INCORPORATING COMPONENT-BASED DESIGN IN THE CATEGORY-THEORETIC FRAMEWORK FOR COMPOSITION OF FAULT-TOLERANT SOFTWARE

Anil Hanumantharaya

A Thesis

In

The Department

Of

Electrical & Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science
in Electrical & Computer Engineering
at Concordia University
Montréal, Québec, Canada

August 2003

Your file Votre référence
ISBN: 0-612-83866-8
Our file Notre référence
ISBN: 0-612-83866-8

Abstract

Incorporating Component-Based Design in the
Category-Theoretic Framework for
Composition of Fault-Tolerant Software

Anil Hanumantharaya

With the increasing use of software in many systems like telecommunications, e-commerce, manufacturing, etc., and the need for reliable services in these systems, there is an ever-growing demand for providing fault-tolerance. Generally, software is built without concentrating much on the fault-tolerant aspect, and fault-tolerance is typically an additional feature to ensure reliability if ever a failure has been encountered. However, there are many legacy software systems that are being deployed in highly critical applications where fault tolerance is inevitable. Various methods have been put forth in the literature for designing fault-tolerance, including a component-based methodology, wherein fault-tolerance is separated from the functionality, and fault-tolerant components, such as correctors and detectors, are added to achieve

the desired reliability. Utilizing the concepts of the component-based design, we propose a category theoretic framework for the composition of these fault-tolerant components with a fault-intolerant program. We illustrate our proposed approach to compose the fault-tolerant components with a fault-intolerant program to result in a final fault-tolerant program through two case studies. In our first case study, we show the feasibility of our approach by composing the fault-tolerant components for a distributed mutual exclusion algorithm using our proposed approach. In the second case study, we decompose the fault-tolerant Label Distribution Protocol and prove the correctness of the design of the fault-tolerant components. Furthermore, the formal specification and verification of these case studies has been conducted using Specware. Some of the benefits of the proposed approach include (a) traceability of all the sorts, operations and properties used to derive the composed program, (b) well-defined interfaces, that allows components to interact in a well-specified behaviour, and (c) reuse of specification for subsequent similar system design.

Acknowledgments

I am indebted to my supervisors, Dr. Purnendu Sinha and Dr. Anjali Agarwal, for their guidance and constant support all through the course of this work. I would like to acknowledge with thanks the financial support I received through my supervisors. I would also like to thank Dr. Sinha for rescuing me when the maths seemed daunting.

I would also like to express my gratitude to my parents, brother and sister, for their love, affection and confidence throughout the duration of my study in Montreal.

My sincere thanks goes to Professor Sandeep Kulkarni for answering all my questions regardless of how trivial they were. I would like to take this opportunity to thank Dr. Kulkarni for giving me valuable insights and suggestions, which helped in my understanding of the concepts of *Detectors* and *Correctors*. I would also like to thank Mr. Adrian Farell for helping me understand the fault-tolerance additions in the Label Distribution Protocol.

Finally, I thank all my friends in the Research group for many intellectual discussions that we had for the past year. Specifically, I would like to thank Vasudevan for helping me get a grip on Category Theory and all its abstract concepts.

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we illustrate the issues related to the design of software fault-tolerance, and present an orderly way of designing fault-tolerant programs. We begin by addressing the need for fault-tolerance in software, followed by a brief summary of the current approaches to developing fault-tolerant software and their limitations. We then explore some of the industrial approaches to composing software components followed by the component-based approach to achieve fault-tolerance. Finally we provide a short summary of the category-theoretic approach for composition, followed by the contribution and an outline of this thesis.

1.1 Need for Fault-Tolerance in Software

Software has pervaded every facet of our society, affecting our everyday life in a direct or indirect way, by influencing everything that we depend on, like, for example,

transportation, manufacturing, telecommunications, shopping, and many others. The ever-increasing demand for ways of making every-day life easier has led to new and better products and trying to improvise them has lead to further exploration in the area of software systems. Many of these modern day utilities, which we take for granted, would not have been possible without these continually evolving software systems.

Although software is being increasingly used, it is far from perfect. An increasing number of today's computer systems consist of an intricate connection of hardware and software components that are prone to malfunction or failure. The failure of such components could lead to unforeseen and potentially disruptive failure behaviour, eventually resulting in service unavailability in many systems, or lead to other catastrophic scenarios. It is expected that these systems continue to function even in the presence of failures of the underlying computing platform. The escalating demand on such well-defined and well-functioning computer systems has led to an increasing demand for such dependable systems, which provide reliability by handling faults in a complex computing environment [16]. Thus, with the growing use of software in a variety of areas that include telecommunications, air and ground transportation, defence, the need to provide continuous service in these areas has led to an increasing demand for fault tolerance in software.

A fault-tolerant system can thus be defined as a system that has the capability to recover from, or in some way tolerate faults and continue with its normal oper-

ation. This means that the system continues to function delivering a desired level of functionality, even when exposed to an adverse environment. Developing systems with such fault-tolerant characteristics is thus becoming increasingly important. Because of these requirements, the software for such fault-tolerant systems is generally large and complex. The design of such complex fault-tolerant systems and the subsequent correctness establishment to ensure that they continually provide the desired functionality despite the presence of faults tends to be a tedious effort.

The lack of techniques to ensure that software designs are free from design faults has also led to the use of fault tolerance techniques as an additional layer of protection. Software fault tolerance is the use of such techniques to facilitate the sustained delivery of services at an acceptable level of performance and safety, following the activation of a design fault.

1.2 Existing Approaches to Fault-Tolerance and their Limitations

There have been several methods proposed in the literature for designing fault-tolerance. Some of these methods by which fault tolerance can be achieved are replication [27], state machine approach [30], and checkpointing and recovery [21]. A brief explanation of each of these methods follows: Checkpointing can be considered as one of the early works in converting a Fault Intolerant Program (FIP) to tolerate

faults, and has primarily focused on correction mechanism [21]. Checkpointing involves periodic saving of the state information that is being considered as a consistent state of the system. Once the fault causes a system failure, the system is reinitialized and resumes operation from that saved, valid state. The fault detection mechanism in this case is usually provided by the hardware, which sets a *boolean* sort in the event of a fault. A recovery procedure is then initiated on fault detection to handle the fault. This approach assumes that software defects cause system failures only if specific inputs occur with a particular timing. These defects are the most difficult to find and repair during development, and the majority of software defects that remain following the testing and deployment of a system are of this type [23]. In the state machine approach, replication is used to achieve the desired goal of fault-tolerance. In this method, crucial server processes are replicated, and responding to a request involves establishing coordination among the processes as to what the response should be. This approach deals with only fail-stop and Byzantine failures. The disadvantage with the replication-based approach, which is also used in the state-machine, is that replication requires operations to be deterministic on the replicas, meaning that the response depends on the initial state and any previous operations performed. Many a times, fault-tolerant systems have to be modified so that they can deal with new types of faults that were not taken into account in the earlier design.

Liu [22] presents an approach for specifying and verifying fault-tolerant, real-time programs by transformations. When a fault occurs, the redundancy operations that

help the system to recover to an error-free state results in an increase in the set of transitions as compared to an intolerant program. In this approach, refinement is used to obtain a program tolerant to faults from a FIP, and a number of refinement conditions for fault tolerant verification are provided.

Another way of achieving software fault tolerance is by the use of software wrappers. A wrapper is a piece of software enclosed around another component limiting what that component can extend to other components or its environment. It achieves this in a non-invasive way by affecting the functionality of the component without modifying the component's source code [35]. Wrappers monitor the flow of information into and out of the component, and try to keep undesirable values from being propagated. It is possible that any illegal output generated by an off-the-shelf component that is not anticipated (not present in the component specification) may not get detected by the wrapper, since the detection technique is based only on anticipated faults.

1.3 Industrial Perspectives to Compose Software Components

Component Based Software Engineering (CBSE), the branch of software engineering devoted to exploring and applying this paradigm of component-based software is thus based on the premise of reusing components so that recurring actions need

not be written repeatedly. The difficulty in reuse of software is due to fact that dependencies that are not so obvious cannot be described explicitly [13]. Some of the component infrastructure technologies currently used in CBSE are EJB, CORBA, and COM/DCOM. The component-based approach diminishes the cost of software development, allowing systems to be assembled rapidly, and reduces the spiralling maintenance burden associated with the support and upgrade of large systems. Some of the advantages of CBSE approach thus include: less burden of reprogramming commonly used operations, software reuse, and reduction in the time to market software. On the other hand, identifying a component that suits the application needs is a difficult task. Also reliability on unknown, third party black-box components increases.

1.4 Role of Formal Methods

A major obstacle currently being faced in the development of software intensive systems result from the system and software specification being far from adequate. The requirement documents, which are the starting point for any software process usually, define much of the functionality of the software system under construction, but most of the time, many of the details, which should have been expressed, clarified and solidified, are not tackled. The consequence of these actions is that there are inconsistencies in the subsequent phases of design and coding, which may not be noticed till a later stage. Rectifying these specification flaws detected at later stages in the

life-cycle is much harder than if the flaw were detected and fixed at the specification stage [36]. This has resulted in a strong thrust to produce more precise and consistent specifications.

The software engineering literature is filled with declarations on how to develop software, for example, "Object-oriented development is the best way to obtain reusable software". Most of the time, these declarations are not augmented with either logical or experimental evidence. This leads to the conclusion that much of software engineering is based on a combination of anecdotal experience and human authority [17]. Practitioners have begun to investigate fully the foundations on which software engineering is based leading to research activities in the formal methods area. Formal methods are the applied mathematics of computer systems engineering. The mathematics of formal methods includes predicate calculus, recursive function theory, lambda calculus, programming language semantics, and discrete mathematics. To this mathematical base, formal methods add notions from programming languages such as data types, module structure, and generics.

Currently, many software development methods use Data Flow Diagrams, Finite State Machines, and Entity-Relationship Diagrams among other things to aid the software engineer to develop better specifications. Wood [36] identifies the drawbacks in these methods which include lack of precise detailed description of the specification, inability in reasoning about the specification, a smooth transition to developing a design and implementation, and mentions how formal specification methods can over-

come these basic objections. The reason being formal methods are detailed enough, allow a specification to be reasoned about, and have a smooth path from specification through design to implementation. Furthermore, [36] cites the successful application of some of the formal methods to real-life, complex software system specifications, like the CICS work at IBM [26], and the oscilloscope work at Tectronix [12], as some of the reasons why formal methods should be considered for software system specification.

A specification represents the desired behaviour of the program. It consists of a set of properties that are to be proved about the implementation model. The properties in the specification need not be a complete specification of the desired behaviour but the general behaviour of the system, for example, that a protocol doesn't deadlock, or that it guarantees mutual exclusion [24].

1.5 Observations and Suggestions

To overcome the limitations of the existing approaches to designing fault tolerant software, Arora and Kulkarni [4, 5] proposed a component-based approach, where the fault-tolerant mechanisms are developed as components. Thus to enhance an existing fault-tolerant solution, a new fault-tolerant component is composed with it to tolerate a new fault type. They also show that these components do not interfere with the original program thereby reducing the risk of introducing new faults. In [19], Kulkarni formulates that a fault tolerant program can be separated into a basic fault intolerant program and a set of fault-tolerant components, namely, detectors and

correctors. The detector ensures that the safety predicate holds for the system state and the corrector design depends on the correction predicate that is to be enforced on the system state. Kulkarni [19] also shows that the use of detectors and correctors is more general and that existing methods can be alternatively designed in terms of detectors and correctors. There have been several approaches proposed in the literature [5, 21, 22, 30] for achieving fault-tolerance, however, as per our observations, there is a lack of rigorous methods to compose fault-tolerant software. This thesis proposes a complementary approach for composing components based on category-theoretic concepts. The advantage of our approach is that there is a traceability of all the sorts and axioms used to derive the composed program, the interfaces are well defined, and the specifications can be subsequently reused.

The reason for following the component-based approach is that the current trend in software development is to build software systems as components, where the component is a non-trivial, nearly independent, and replaceable part of a system that has a clear function in the context of a well-defined architecture [8]. Component-based software development focuses on constructing complex software systems using existing software components. In the initial days of software when storage (memory) was a major factor, programmers came up with the concept of subroutines, which allowed them to execute repeated actions by placing them in a subroutine, and invoking the subroutine in place of those segments thereby saving space. This can be considered as the beginning of software reuse. Subsequently software libraries came into being,

wherein a collection of commonly used routines were bundled together. These library routines could have even been written by other programmers and the user would invoke it just as it was another piece of code. The subroutine can thus be seen as an atomic statement, i.e., a component, wherein the programmer is not concerned with what algorithms or data structures have been used to write the subroutine, its development history or about its storage management and so forth [10]. Just as subroutines freed the programmers from worrying about the details, Component-Based Software Development (CBSD) places emphasis on composing software systems rather than writing (programming) software all over again. The basic foundation of this approach is that common functionality that reappears with sufficient regularity should be written once, and these common systems should be assembled through reuse rather than rewriting over and over again. Another advantage is that composing larger systems from smaller systems, which have been shown to be correct, helps in achieving a higher degree of confidence in the correctness of the overall system. Furthermore not having to recreate parts that others have already built means the user ends up doing lesser work [7].

1.6 Proposed Category Theoretic Framework

We now give a brief overview of our proposed category theoretic framework. The categorical framework has a well-suited structure for combining components. In this framework, the interactions between objects are described by the morphisms of the

category. We have adapted the calculus of modules proposed in [14], which defines a means of describing modules and modularity concepts like encapsulation or genericity. Their approach is expressive and modular, expressive because of the use of logic, which is necessary to specify complex systems. Modularity is maintained by structuring the description of the system in terms of modules that allows very precise description. There are three levels of description: [25]

- System Level : a system is described by modules that are interconnected by morphisms and on which operations can be performed,

- Module Level : a module is composed of four specifications linked by specification morphisms,

- Specification Level : each specification is a logical theory consisting of a signature that gives the vocabulary, i.e., attributes and methods and a set of formulae to describe the behaviour.

We illustrate the applicability of our approach with two case studies. The verification of our approach is done using a theorem prover. A theorem prover is based on logic, which is a set of axioms and inference rules based on which it is possible to reason from premises to conclusion, thus proving the logic, which is usually a mathematical proposition. Using a theorem prover, we can reason at an abstract level without going into the details about a system design before certain parameters have been fixed (for example, an internet address or a process identifier).

1.7 Contribution

The aim of this work is to apply the concepts of category theory for the composition of a FIP with fault-tolerant components leading to an overall fault-tolerant program. Our specific contributions in this thesis are: we

1. apply the concepts of category theory for the composition of detector and corrector components with a FIP,
2. decompose a fault-tolerant multimedia protocol into a fault-intolerant program and detector and corrector components,
3. establish proof of correctness of detector and corrector components,
4. provide a rigorous and consistent composition of fault-intolerant program and fault-tolerant components utilizing the category theoretic framework, and finally
5. realize the proposed composition approach, i.e., perform compositional specification and verification through *Specware*, a mechanized formal tool from the Kestrel Institute.

1.8 Outline of the Thesis

We proceed as follows: In Chapter 2, we give formal definitions of a program, problem specification, fault, and fault-tolerance and explain the approach of [4, 5] for designing the fault-tolerant components. This is followed by an introduction to our proposed category-theory based approach for composing software components. Chap-

ter 3 describes the first case study involving a distributed mutual exclusion program with relevant failure scenarios. Chapter 4 describes the second case study involving the fault-tolerant version of the label distribution protocol. In these case studies, after having identified the fault-intolerant program and fault-tolerant components, we illustrate how concepts of category theory can be used to compose these software components to derive a fault-tolerant program. The formal specification of the composed module for the second case study, and its verification is presented in the Appendix. Finally Chapter 5 concludes with discussions and future research directions.

Chapter 2

Preliminaries

In this Chapter, we present some of the definitions adapted from [19] and the theory behind [4, 5] approach for designing fault-tolerant components and their composition with the fault intolerant program. We then proceed to explain our category-theoretic formal framework for composing these components with the FIP. Finally, we provide a brief description of the *SPECWARE* tool that we have used for the purpose of specifying and verifying the composition.

2.1 Definitions

We present some of the definitions that have been adapted from [19] for describing the component-based approach of Kulkarni. In their work, a program is intuitively denoted by state transitions, and a problem specification is indicated as a set of state sequences. Based on the work of Arora and Gouda [3], faults are depicted as state

transitions which allows the representation of a rich class of faults followed by the definition of fault-tolerance. Finally we list the definitions of detectors and correctors and their design methodology.

Program :

Based on the work of Chandy and Mishra [9], a program, p is defined as a finite set of actions over a set of variables. Each of these variables' has a value from a predefined non-empty domain. An action has the form:

$< name > :: < guard > \rightarrow < statement >$, where *name* is the action name and *guard* is a boolean expression over the variables of p.

State :

A state of p is defined by a value for each variable of p, chosen from the predefined non-empty domain of the variable's value.

State Predicate :

A state predicate of p is a boolean expression over the variables of p.

Invariant :

An invariant of p is a logical expression whose truth value is always *true* whenever the program executes correctly [11], i.e., it is a predicate that characterizes the set of fault-free states reachable during the fault-free execution of p.

Problem Specification :

A problem specification is a set of sequence of states that is suffix and fusion closed. Suffix closed means that if a state sequence σ is in that set, then so are all its suffixes.

If α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and x is a program state, then fusion closed means that if $< \alpha, x, \gamma >$ and $< \beta, x, \delta >$ are in that set, then so are the state sequences $< \beta, x, \gamma >$ and $< \alpha, x, \delta >$.

Closure :

A state predicate R is preserved for an action of p, if starting from any state of p where R holds, and the guard of the action is *true*, executing the statement of the action yields a state where R holds. R is closed in p *iff* each action of p preserves R.

Fault Actions :

A fault can be represented by actions, called fault actions whose execution disturbs the program state.

$Fault - span$:

Consider a set of fault actions F that a program is subject to. The set of states that p reaches when it executes in the presence of actions in F is called fault-span predicate of p with respect to F. It also holds at each fault-free state of p.

$Fault - Tolerance$:

During the normal execution of a program p, starting from any state where the invariant S holds, an action of p results in a state where S continues to hold, whereas executing an action from F, the set of fault actions, may result in a state where S does not hold. However, in this state, T, the fault-span holds, subsequent actions of p and F result in states where T holds and when actions in F stop executing, subsequent execution of actions of p alone eventually yield a state where S holds,

from which point normal execution is restored.

From the above definition it can be implied that if T, the fault-span is S itself, then p is masking F-tolerant, else if $T \neq S$, p is non-masking F-tolerant.

2.2 Detectors

A Detector is defined as a component that ensures that the safety property of the distributed system is preserved. It does this by detecting if the safety predicate is satisfied by the system state.

Detectors form the basis of fail-safe tolerant programs. To validate this statement [19] has shown that detectors are sufficient to design fail-safe tolerant programs and that detectors are necessary to design fail-safe fault-tolerance. Some of the techniques used to design fail-safe tolerant programs are comparators, error detection codes, consistency checkers, watchdog programs, alarms, exception conditions, which are all instances of detectors.

2.2.1 Designing Detectors to achieve fail-safe tolerance

The detector component ensures that the program does not deviate from its normal execution and that any bad state is detected by this component. Detectors are designed with a given detection predicate.

2.3 Correctors

A Corrector is defined as a component that ensures that the liveness property of the distributed system is preserved. It does this by detecting if the safety predicate is satisfied by the system state and also correcting the system state in order to satisfy that predicate whenever it is not satisfied.

Similarly [19] shows that correctors form the basis of nonmasking tolerant programs. To validate this statement, [19] has shown that correctors are sufficient to design non-masking tolerant programs and that correctors are necessary to design non-masking fault-tolerance. Some of the techniques used to design nonmasking tolerant programs are voters, reset procedures, rollback recovery, rollforward recovery, which are all instances of correctors.

2.3.1 Designing Correctors to achieve non-masking tolerance

Arora [2] proposes a method for designing non-masking fault tolerance based on which the corrector component is designed. It puts forth the following steps to design a program that satisfies a given problem specification and is non-masking tolerant to a given set of fault actions:

1. Designing a fault-span T, in which a state predicate T is constructed which is weak enough so that the fault actions preserve it.
2. Designing the invariant S, in which a state predicate S is constructed which is strong enough so that the safety properties of the problem specification are

met.

3. Designing program actions that achieve non-masking tolerance by ensuring that T converges to S, meaning that each of these actions preserve T as well as S.
4. Designing program actions that satisfy the problem specification, which ensures that the program specification is satisfied in all computations that start from states where S holds.

2.4 A Formal Approach to Putting Components Together

Algebraic specification deals with modelling system behaviour using algebras (a collection of values and operations on those values) and axioms that characterize algebra behaviour, and composing smaller specifications to result in larger specifications. The specification building operations defined by category theory constructs are used for the composition operation. In this section we give an overview of component (module) composition utilizing the concepts of category theory. We begin with the definitions of some of the terms in category theory that are relevant for our work.

Category theory is an abstract mathematical theory used to describe the external structure of various mathematical systems [32] and is used in this thesis to describe the relationship between specifications.

A category consists of a collection of $C-objects$ and $C-arrows$ between objects

that respect the following properties:

1. There is a $C - arrow$ from each object to itself. This is called the *identity* morphism.
2. $C - arrows$ are composable, i.e., each morphism is associated with an object A that is its domain and an object B that is its co-domain.
3. Arrow composition is associative.

- **Signature:** The structure of an algebraic specification is defined in terms of sorts, an abstract collection of values, and operations over those sorts. This structure is called a *signature.*
- **Specification:** A signature defines only the syntax of a solution. The semantics are specified by extending the signature with axioms defining the intended semantics of the signature operations. This extension of the signature with associated axioms is called a *specification.*
- **Specification Morphism:** A *specification morphism* is a pair of functions that maps sorts and operations from one specification to compatible sorts and operations of a second specification such that a) axioms of the first specification are theorems in the second specification and b) source operations are translated compatibly to target operations. Intuitively, specification morphism defines how one specification is embedded in another.
- **Colimit:** The *colimit* operation takes a *specification diagram* as input and produces a specification called the colimit of the diagram. In this operation, the

morphisms between the specifications have to be first indicated. A *diagram of specifications* is a directed multi-graph whose nodes are labelled with specifications and whose arcs are labelled with morphisms. The colimit operation is then applied to a diagram of specifications linked by morphisms. The colimit contains all the elements of the specifications in the diagram, but only elements that are linked by arcs in the diagram are identified in the colimit. For example, any diagram containing objects A_i and A_j, and morphism a_x, the *colimit* is an object L and a family of morphisms I_i, I_j such that for each $I_i : A_i \rightarrow L$, $I_j : A_j \rightarrow L$, and $a_x : A_i \rightarrow A_j$, then $a_x \circ I_j = I_i$.

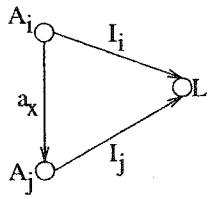

Figure 2.1: Colimit Function

Module specifications are defined by utilizing the notion of *push–out* operation from category theory. Given specifications A and B, and a specification R describing syntactic and semantic requirements along with two morphisms f and g, the *push–out* operation gives the specification P which contains A and B.

Formally, given specification morphism $f : R \rightarrow A$ and $g : R \rightarrow B$, a specification P together with specification morphisms $h : A \rightarrow P$ and $k : B \rightarrow P$ is called the *push–out* (*of* f *and* g), provided that the module commutes, i.e., $h \circ f = k \circ g$, where $\circ$ denotes composition.

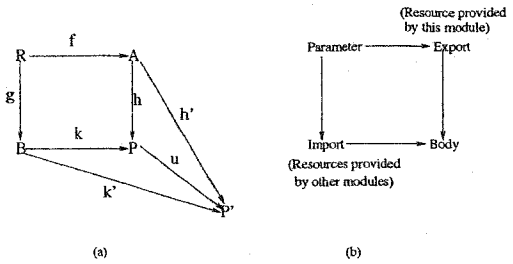

Figure 2.2: (a) Push–out and (b) Module Interfaces

2.4.1 Module Interfaces

Modules are expressed in terms of externally visible interfaces defining what services a module provides and requires in order to be used in a system. We express these module interactions by the *export* and *import* interfaces that are essentially specifications, and their sorts and operations linked together via spec-morphisms.

A module specification $MOD = (PAR, EXP, IMP, BOD, f, h, g, k)$ consists of four specifications (a) *parameter* specification (PAR), (b) *export* interface specification (EXP), (c) *import* interface specification (IMP) and (d) *body* specification (BOD), and and four mapping morphisms f, h, g, and k such that the following diagram commutes, i.e., $f \circ h = g \circ k$ (Figure 2.2(b)).

2.4.2 Composing Modules Together

Composition: To capture module interactions, our proposed composition scheme allows two modules to be interconnected via the export and import interfaces. Their *push-out* is the resulting specification of the composed module. Figure 2.3 depicts the composition operation. In the figure, *Module1* has four objects namely Parameter

(R_1), Export (A_1), Import (B_1) and the pushout of these 3 objects giving the Body (P_1) which is the specification of *Module1*. Similarly *Module2* has R_2, A_2, B_2 and P_2 as its Parameter, Export, Import and Body respectively. In Figure 2.3(a), *Module1* imports via specification B_1 whatever *Module2* exports via specification A_2. The compatibility of the parameters (or semantic constraints) is governed by the morphism t. Furthermore, the following property must be respected: $g_1 \circ s = t \circ f_2$. Basically, in category-theoretic terms, Modules 1 and 2 are diagrams that commute individually by the specification morphism relationship of $f_1 \circ h_1 = g_1 \circ k_1$ and $f_2 \circ h_2 = g_2 \circ k_2$ and their colimit would produce the required composed module, which would now commute by the relationship $f_1 \circ h_1 \circ m_2 = t \circ g_2 \circ k_2 \circ m_1$ as in Figure 2.3(b). Since the composed module also commutes, its specification is proved correct thereby helping in the reusability of the module.

The module (P_{12}) obtained by composing the two sub-modules (1 and 2) is also a diagram with its Parameter as the parameter of *Module1*, Export as the export of *Module1*, Import as the import of *Module2* and the Body as the union of bodies P_1 and P_2 over the export of *Module2*. In this case, the resulting composed module P_{12} is *(R_1, A_1, B_2, P_{12})*, where P_{12} is the *push-out* of P_1 and P_2 over B_1(Figure 2.3(b)).

2.4.3 Interconnecting Modules via the Union Operation

Many a times, two specifications derived from a common specification need to be combined. The desired specification after combining should consist of parts unique to

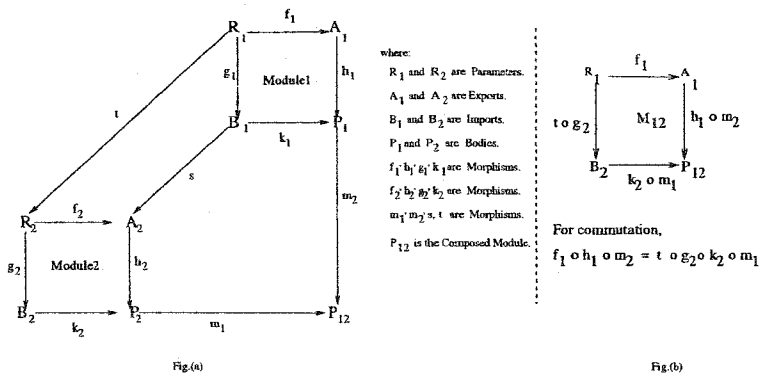

Figure 2.3: (a) Composition of Two Modules and (b) Composed Module

the two specifications and some "shared part" that is common to both. As illustrated in [14], the union operation in category theory provides an interconnection mechanism between module specifications. Instead of taking the set theoretical union, the shared part is designated, and the union construction thus identifies this shared subpart of both module specifications and the disjoint union for all other parts. The connection between the shared submodule and the other module specifications is given by a pair of module specification morphisms. Given module specifications MOD_j for $j = 0$, 1, 2 and module specification morphisms $f1 : MOD0 \rightarrow MOD1$ and $f2 : MOD0 \rightarrow MOD2$, the union $MOD3$ of $MOD1$ and $MOD2$ via $MOD0$ and $f1$, $f2$, written as: $MOD3 = MOD1 +_{(MOD0,f1,f2)} MOD2$ is given by the construction below, and shown in the Figure 2.4.

1. $PAR3 = PAR1 +_{PAR0} PAR2$
2. $EXP3 = EXP1 +_{EXP0} EXP2$
3. $IMP3 = IMP1 +_{IMP0} IMP2$
4. $BOD3 = BOD1 +_{BOD0} BOD2$

The definitions of f_3, g_3, k_3 and h_3 make use of the fact that $f1$ and $f2$ are module specification morphisms such that all subdiagrams, except the diagram having f_3, g_3, k_3 and h_3 are already commutative. Hence the property to be respected to show the correctness of the union construction is to prove that

$PAR_j \rightarrow PAR3 \rightarrow EXP3 = PAR_j \rightarrow EXP_j \rightarrow EXP3$ for $j = 1, 2$.

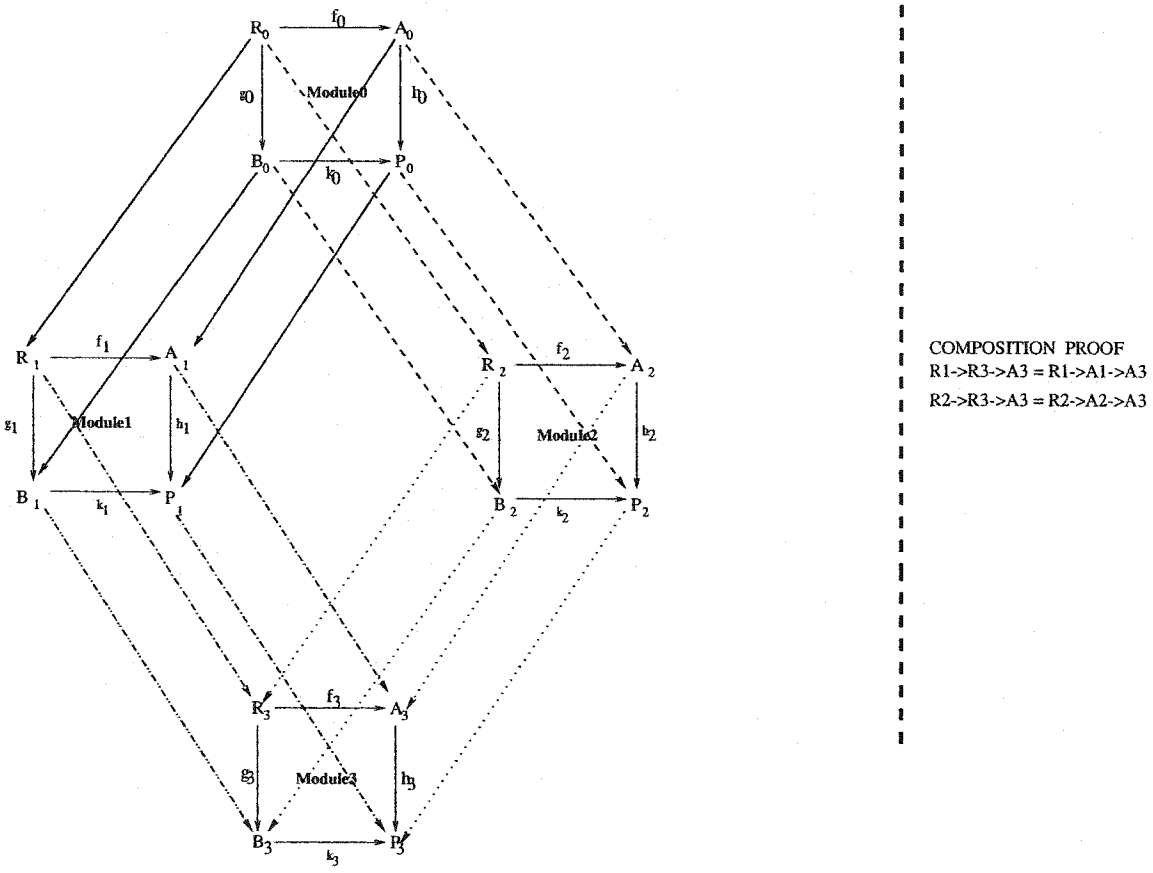

Figure 2.4: Union Operation

All the above concepts are expressed and implemented in *MetaSlang*, the language used in Specware.

2.5 Our Framework for Composing Software Components Together

The specific goal of the proposed framework is to facilitate the construction of a dependable program by composing a fault-intolerant program with fault-tolerant components. The framework adapts the approach presented in [5] for transforming a fault-intolerant program into a fault-tolerant one. The framework for our approach is shown in Figure 2.5.

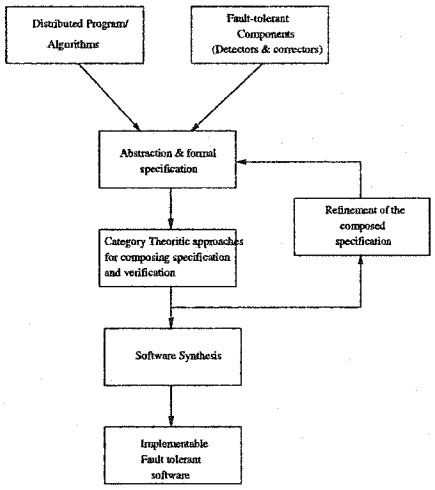

Figure 2.5: Framework for composition

We first identify the failure scenarios of the algorithm or the fault-intolerant program under study and construct the appropriate fault-tolerant components following the guidelines proposed in [5]. The program along with the components are then specified in an abstract formal specification and then subsequently verified. After having specified and verified these individual software components, we then utilize concepts

of category theory for composing the program and its fault-tolerant components to result in a single composed program that is capable of handling specific types of fault. The resulting composed specification of the fault-tolerant program is then verified to ensure that the program satisfies that desired dependability requirements. Note that the framework also allows for further refinement of the composed as well as individual component specification. This enables enriching the specification with additional details in terms of data structures and operations. Each refinement step is subsequently subjected to verification to ensure that there are no inconsistencies. Through software synthesis, for the given level of abstraction, we can obtain an implementable code that would be fault-tolerant by the virtue of correct-by-construction approach. The proposed approach has been realized with the support of Specware [33] tool from Kestrel Institute. Once the specification has been refined into a suitable form, Specware supports standard library components to be converted into components in some executable programming language using a code generator.

2.6 Specware

In this section, we provide a brief overview of the Specware tool, which has been used to specify and verify the concepts of our approach.

Specware is a system which aims to provide a formal support for specification and development of software [34]. It allows users to precisely specify the desired functionality of their applications and to generate provably correct code based on

these requirements. The foundations of Specware are category theory, sheaf theory, algebraic specification and general logic. Using Specware, one can construct formal specifications modularly and refine such specifications into executable code through progressive refinement. The software development in Specware is characterized by two tenets [34]:

- **Description:** A description is a collection of properties that we ultimately wish to build. These descriptions are progressively refined by adding more properties, till a model is reached which satisfies these properties.
- **Composition:** Complexity and scale are handled by utilizing the composition operators which allow bigger descriptions to be put together from smaller ones. The colimit operation from category theory is pervasively used for composing structures of various kinds in Specware.

Adapting from the introduction presented in citemanual, the following discussion describes Specware briefly. Specware aids in expressing requirements as formal specifications without regard to the ultimate implementation or target language. Specifications describe the desired functionality of a program independently of such implementation concerns as architecture, algorithms, data structures, and efficiency. This makes it possible to focus on the correctness, which is crucial to the reliability of large software systems. Using Specware, the analysis of the problem can be kept separate from the implementation process, and implementation choices can be introduced piecemeal, making it easier to backtrack or explore alternatives.

Chapter 3

Case Study 1: The Mutual Exclusion Algorithm

In Chapter 2, we introduced the basics for the construction of fault-tolerant components and our category-theoritic based approach for modular composition. In this chapter, we discuss the application of our approach with a case study based on the fault-intolerant mutual exclusion program of Raymond [28] and Snepscheut [31]. This has been subsequently used in [5] to introduce an approach to develop a fault-tolerant program utilizing corrector and detector components. We proceed as follows: First we briefly describe the mutual exclusion problem and the faulty scenarios that may arise. This is followed by a description of the actions of corrector and detector components. We then illustrate the composition of these components using category theory, followed by their formal specification and verification using Specware.

3.1 Brief Description of the Mutual Exclusion Algorithm

Consider a set of N processes executing concurrently with each process having critical sections to be executed. The conditions to be satisfied before a process executes a critical section are:

1. There must be only one process executing the critical section at any time. A process can enter the critical section only if it possesses the token.

2. The algorithm should be non-blocking, i.e., when the process is in a critical section, it never halts and if it does in a non-critical section, it does not interfere with other processes.

3. There should be no deadlock, i.e., if a process tries to enter the critical section, eventually it succeeds.

The algorithm as such has no fault-tolerant features. To make the algorithm fault-tolerant, [5] gives details of the composition of a corrector and a detector after identifying the fault scenarios. Fault tolerance is needed to ensure that there are no inconsistencies in the execution of the critical section and to ensure that at any time, only one process executes the critical section.

Faults arise when one of the processes in the directed-tree fail-stops, which in turn leads to inconsistencies in the parent and holder relationship (will be explained

shortly). If the failed process also had the token, then a new token must be generated. The detector should ensure that such faults are detected and the corrector ensures that the program not only detects but also recovers to a safe state on the detection of such faults.

The algorithm is based on the token approach, wherein a unique token is circulated between the processes, and entering a critical section is subject to a process having the token. The following are the assumptions made in this algorithm.

1. All the processes are organized in a tree pattern.

2. Each process maintains a request queue, which stores the requests of those neighbouring sites that have sent a request to the process, but have not yet been sent the token.

3.2 Fault-Intolerant Program (FIP)

The normal working of the FIP consists of the following actions:

1. The program either makes or propagates a request to the holder process for getting the token.

2. The program transmits the token to fulfill a pending request from the neighbouring process.

3. The program accesses the critical section when it has the token.

The program includes the following sorts which are the basic data structures of the FIP module:

- *holder.x* : holder for the node x, indicating the location of the token relative to the node x itself.
- *parent.x* : parent of the node x, in the tree.
- *using* : indicates if the node has token or not (i.e., true or false).

The FIP module has the following operations:

- *REQUEST-CS:* This operation requests for a critical section by sending a request message to the holder process. *REQUEST-CS* acts on the *holder.x* variable and the request queue. It performs the following actions:

  1. To enter a critical section, the process sends a request to its holder process, provided the token is not with the process itself and its queue is empty.
  2. It then adds its own request to its request queue.

- *RECEIVE-REQ:* This operation does the processing of the Request message received. $RECEIVE - REQ$ operation acts on the *holder.x* variable and the request queue. It performs the following action:

  1. When the holder (i.e., the process in the path) receives this message and it is not the process that has the token, it places the request message in

the request queue and sends a request message along the path, i.e., to its holder. This is done if it has not already sent a request on its outgoing edge (to its holder).

- *FORWARD-TOKEN:* This operation acts on the token variable and the *holder.x* variable and is invoked after completion of execution of the critical section. It performs the following actions:

  1. It checks if the request queue is empty.
  2. If the queue is empty, it holds the token till it receives a request.
  3. If it is not empty, the entry at the top is removed and the *token* is forwarded to that process.
  4. It then sets its *holder.x* to that process.

- *ENTER-CS:* This operation is invoked when the process has to enter the critical section. It acts on the token and request queue.

  1. A process enters the critical section if it receives the token and its own entry is at the top of its request queue.
  2. The entry is deleted from the request queue and the process enters the critical section.

3.3 Considering Fault Scenarios

In our discussion, the failure of a process is assumed to be fail-stop. The failure of a process may lead to partitioning of the tree structure in the FIP leading to inconsistencies in the parent and holder relationship. Furthermore, it may lead to loss of the token being circulated, when the process having the token fail-stops. To mask these failures, we need a corrector that corrects the parent tree, i.e., it reconstructs the parent tree and another corrector that sets right the holder relationship of the reconstructed tree. The consequence of these actions results in the holder relationship being identical to the parent relationship and the root process being the token-holder.

3.3.1 Corrector for Rectifying the Parent Relation (C1)

When there is a failure in one of the nodes, it leads to program states where there are multiple trees (because of the broken link) and unrooted trees [5]. We therefore need a corrector that rectifies the parent tree in the event of a failure of one of the nodes. The corrector merges the trees and eventually causes convergence to a state where there is only one root for the entire tree. The corrector has the following sorts:

- *colour.j* : colour of the node j.
- *holder.j* and *parent.j* are imported from the FIP module.

The corrector has the following operations:

- *CORRECT-UNROOTED (NT1):* This operation involves the *holder.x* and *parent.x* sorts and has the following steps:

  1. When a node x detects that its *parent.x* is down or its *colour.x* is red, it sets its colour to red.

  2. When the leaf node gets coloured red, it separates from the tree and resets its *colour.x* to green.

  3. It then sets its *holder.x* and *parent.x* to $SELF$.

- *CHECK-COLOUR (NT2):* This operation checks if the parent node is down or its *colour.x* is red.

- *MERGE (NT3):* This operation involves modifications to the *holder.x* and *parent.x* sorts and has the following steps:

  1. Node j merges into the tree of a neighbouring node k when root of j is less than root of k.

  2. After merging, j sets root of j to root of k, $parent.j = k$ and $holder.j = k$.

- *SET-COLOR:* This operation sets the *colour.x* of the node to red.

When the faults stop occurring, if a process is coloured red, then all its children are coloured red. The action $NT1$ of the corrector eventually causes all processes to reach a state where they are coloured green, leading to a state where there are no more unrooted processes. Finally, the graph of the parent relation

forms a rooted spanning tree with the root values of all processes being identical [5]. The export operations after the addition of this corrector are modified versions of *REQUEST-CS, RECEIVE-REQ, FORWARD-TOKEN* and *ENTER-CS* which are named *REQUEST-CS1, RECEIVE-REQ1, FORWARD-TOKEN1* and *ENTER-CS1* respectively.

3.3.2 Corrector for Rectifying the Holder Relation (C2)

The holder relation may still remain inconsistent in one of these two conditions:

1. Holder of j may not be adjacent to j in the parent tree or
2. Holder of j may be adjacent to j in the tree but the holder relation forms a cycle.

This corrector involves addition of predicates to satisfy the liveness property. The actions performed by this corrector are:

1. When the condition (1) holds, C2 sets $holder.x$ to $parent.x$, and
2. When the condition (2) holds and $parent.k = j$, $holder.j{=}k$ and $holder.k{=}j$, C2 sets $holder.j$ to $parent.j$.

With the addition of $C2$, the export operations are enhanced versions of *REQUEST-CS1, RECEIVE-REQ1, FORWARD-TOKEN1* and *ENTER-CS1* which are named *REQUEST-CS2, RECEIVE-REQ2, FORWARD-TOKEN2* and *ENTER-CS2* respectively.

3.3.3 Global Detector (D1)

To enhance the tolerance of the program from non-masking to masking, two detectors are added [5]. The actions of the first detector, explained in this section, affects the state of all nodes, and is hence called the global detector. The second detector, called the local detector since its actions are local to the node, is explained in the next section.

When the corrector actions result in the node setting the holder to itself, it results in the creation of a new token. The safety condition to be satisfied before a new token is generated is that "no other process should have the token".

The addition of the corrector results in a non-masking tolerant program, which eventually converges to a state where the graph of the parent relation is a rooted tree, and the holder of each node is its parent. Therefore, the safety predicate to be detected is to observe if the program is in such a state. To achieve this, the node j initiates a diffusing computation when the $holder.j = SELF$ or the $holder.j = parent.j$, i.e., when node j is the root node. Each node maintains a sequence number which it increments for each diffusing computation that it initiates/propagates. The root proliferates the computation to its children, which in turn propagate the diffusion message to leaf nodes through intermediate nodes. When the leaf node receives the diffusion message, it responds to its parent. The parent in turn, collects the results from all its children and replies to its parent, informing whether a process in the sub-tree has the token or not. The root on receiving the result from all its children,

decides if any node in the sub-tree has the token by inspecting the result. The sorts defined by this module are:

- *phase.j*: phase of the node *j*, can take values *propagating, completed*
- *sequence.j* : Sequence number for node *j*.

The following are the operations of D1:

- *INIT:* This operation performs the following actions.
  1. Increment the *sequence.j*.
  2. Send the diffusion message to all children.
  3. Set *phase.j=propagating*.
- *PROP:*
  1. If *sequence.j* is different from *sequence.*$(Parent.j)$ and j and $Parent.j$ are in the same tree, propagate the diffusion message.
  2. If $holder.j = Parent.j$ and $phase.(Parent.j) = propagating$, return $result = true$.
  3. Returns $result = false$, thus completing the diffusion process.
- *COMPLETE:* This operation is executed at all the nodes.
  1. If all the children have responded with result *true*, all neighbours have propagated the diffusion computation and their result is *true*.

Completion of the diffusion computation with result *true* ensures that the safe predicate is satisfied and the root can generate a new token.

- *ABORT:* This operation completes the diffusion computation prematurely with the result false.

With the addition of $D1$, the export operations are enhanced versions of *REQUEST-CS2, RECEIVE-REQ2, FORWARD-TOKEN2* and *ENTER-CS2* which are named *REQUEST-CS3, RECEIVE-REQ3, FORWARD-TOKEN3* and *ENTER-CS3* respectively.

3.3.4 Local Detector (D2)

The safe predicate to be satisfied when the process is forwarding the token is that it should not be participating in a diffusing computation. The FORWARD-TOKEN operation is therefore enhanced to check that the $phase.j = completed$ and only then is the token forwarded. Similarly the operation ENTER-CS is modified to include a check for $phase.j = completed$ and only then can the process enter the critical section. These two actions are performed by the local detector, D2.

With the addition of $D2$, the export operations are enhanced versions of *REQUEST-CS3, RECEIVE-REQ3, FORWARD-TOKEN3* and *ENTER-CS3* which are named *REQUEST-CS4, RECEIVE-REQ4, FORWARD-TOKEN4* and *ENTER-CS4* respectively.

3.4 Composing the FIP, Corrector and Detector using Category Theory

In this section, we explain the conversion of the FIP into a FTP by composing it with the corrector and detector modules utilizing the concepts of category theory, as illustrated in Section 2.4.

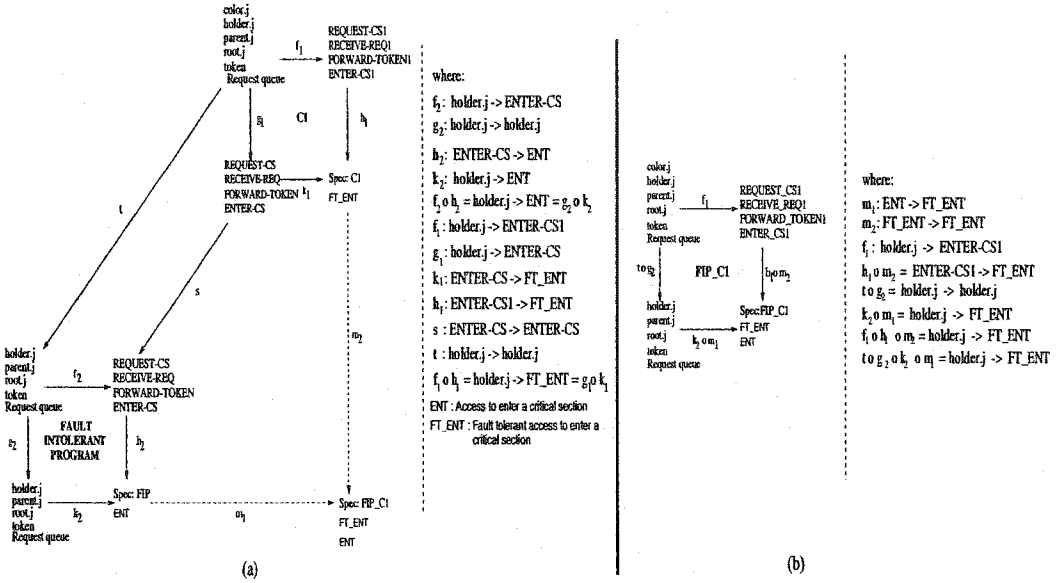

Figure 3.1: (a) Composition of FIP and Corrector 1 (b) Composed diagram of FIP and Corrector 1

3.4.1 Composing FIP and C1

Figure 3.1(a) depicts the individual modules of *FIP* and *C1*. For the composite module *FIP_C1*, composed out of *FIP* and *C1* to exist, it is necessary that these two modules commute individually through the specification morphism relationship as

discussed in Section 2.4[1]. These relationships are shown next to the figure separated by a dashed line.

Each of the modules (components) are depicted in the same way as Figure 2.3 with the four sets of objects as parameters, export, import and body (spec) with corresponding morphisms and mappings. The FIP has the holder, request queue, parent, root and token as its parameters. The properties of the FIP module are: (a) to request access to a critical section (*REQ*), and (b) to enter a critical section (*ENT*). Since *FIP* is the basic module, the import interface and parameter part of the specification are equal: *FIP.import* = *FIP.parameter* [14]. The export entity consists of the operations that achieve the mutual exclusion property, viz., *REQUEST-CS*, *ENTER-CS*, and *RELEASE-CS*. In the Figure 3.1(a), f_2 maps the *holder.j* to the *ENTER-CS* operation, i.e., *ENTER-CS* acts on the variable *holder.j*. The morphism, h_2 maps the operation *ENTER-CS* to the *ENT* property in *Spec:FIP* since the operation *ENTER-CS* provides the property of entering the critical section. The morphism, k_2 maps the variable *holder.j* to the property *ENT*, since this variable is being used in providing the final property of entering the critical section.

The *C1* module has the holder, request queue, parent, using and colour as its parameters. The properties of the *C1* module are: (a) fault-tolerant request to access a critical section (*FT_REQ*), and (b) fault-tolerant access to enter a critical section

[1]At this stage, we feel that pictorial representation of a module and the definition of morphisms placed adjacent to the module diagram would be much more effective. We will present formal specification and verification in Section 3.5.

(*FT_ENT*). The morphism t relates the parameters of the *FIP* with the parameters for *C1*, since *C1* also operates on the same parameters as that of *FIP*. The morphism s maps the operations that are exported by *FIP* module and imported by the *C1* for its proper working. The internal operations of *C1* are private to the *C1* module and as such will not appear in the EXPORT object of the composed module.

By using these specification morphisms (t and s), we can compose the two modules, namely, *FIP* and *C1* resulting in the final composed module of *FIP_C1*. The final specification, *Spec:FIP_C1* has the property of *FIP*, namely, *ENT*, since the morphism m_1 maps the property *ENT* in FIP specification to *FT_ENT* in the final specification. Since the final specification, *Spec:FIP_C1* has the property of *C1*, namely, *FT_ENT*, the morphism m_2 maps the property *FT_ENT* in *C1* specification to *FT_ENT* in the final specification.

Figure 3.1(b) shows the composed diagram (module) of the *FIP* composed with *C1*. It has the properties of both *FIP* and *C1* for (a) entering the critical section via *ENTER_CS1* and (b) masking the fault by merging the tree back when the parent relationship becomes inconsistent via *NT3* operation.

In Section 3.3, we explained the features of the *FTP* and in Section 3.3.1, we identified one of the correctors for the *FIP*. The overall composition of the *FIP* and *C1* provides: re-establishing the parent relationship in the case of a failure of a node/process. With the addition of the *C1*, merging of the trees is done whenever a process goes down, thereby combining multiple trees into one, and the parent re-

lationship among the remaining processes is reconstructed, thus solving the problem of unrooted trees. The final export operations of the composed module are modified versions of the operations exported by the *FIP*, but tolerant to faults that are masked by *C1*.

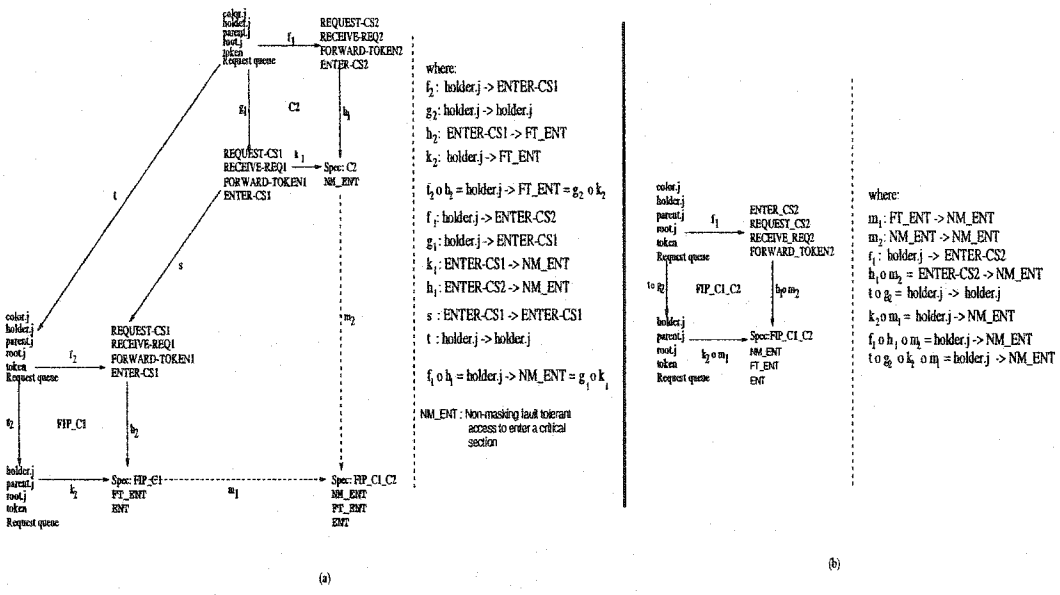

Figure 3.2: (a) Composition of FIP, Correctors 1 and 2 (b) Composed diagram of *FIP_C1_C2*

We have shown how the components, namely, *FIP* and the corrector *C1* can be composed using the concepts of category theory, and the correctness is shown in Figure 3.1(b). The composed module as shown in Figure 3.1(b) also commutes, and hence the combined specification can be reused for subsequent composition.

3.4.2 Composing FIP_C1 and C2

In Figure 3.2(a), f_2 maps *holder.j* to *ENTER-CS1*, h_2 maps *ENTER-CS1* to *Spec:FIP_C1* since it provides property of entering the CS even in the case of failure of the parent

relationship. The morphism, k_2 maps *holder.j* to property *FT_ENT* in *Spec:FIP_C1*, since it is used in providing the functionality of entering a CS in the event of faults. The morphism, g_2 maps *holder.j* to *holder.j* inherited from the *FIP* module. The *C2* module provides the property of non-masking access to enter a critical section (*NM_ENT*). The final specification *Spec:FIP_C1_C2* has the property of *FIP_C1*, namely, *FT_ENT*, since the morphism m_1 maps the property *FT_ENT* in *FIP_C1* specification to *NM_ENT* in the final specification. The morphism m_2 maps the property *NM_ENT* from *C2* specification to *NM_ENT* in the final specification. Thus the composed module shown in Figure 3.2(b) has the property of both *FIP_C1* and *C2* and hence is non-masking tolerant.

At this stage, we have the composite module *FIP_C1_C2* that is capable of tolerating failure in both the parent and holder relationship, when a process in the tree fail-stops. The composed module *FIP_C1_C2* is now tolerant to failure of any of the processes in the tree, but if the process that failed also had the token, then a new token must be generated. A new token should only be generated when no other process has the token. To detect this, the composite module *FIP_C1_C2* is composed with the detector *D1*, which is illustrated in the next section.

3.4.3 Composing FIP_C1_C2 and D1

Figure 3.3(a) illustrates the composition of the *FIP_C1_C2* module and detector module *D1*. The *D*1 module extends the support of non-masking access to enter a critical

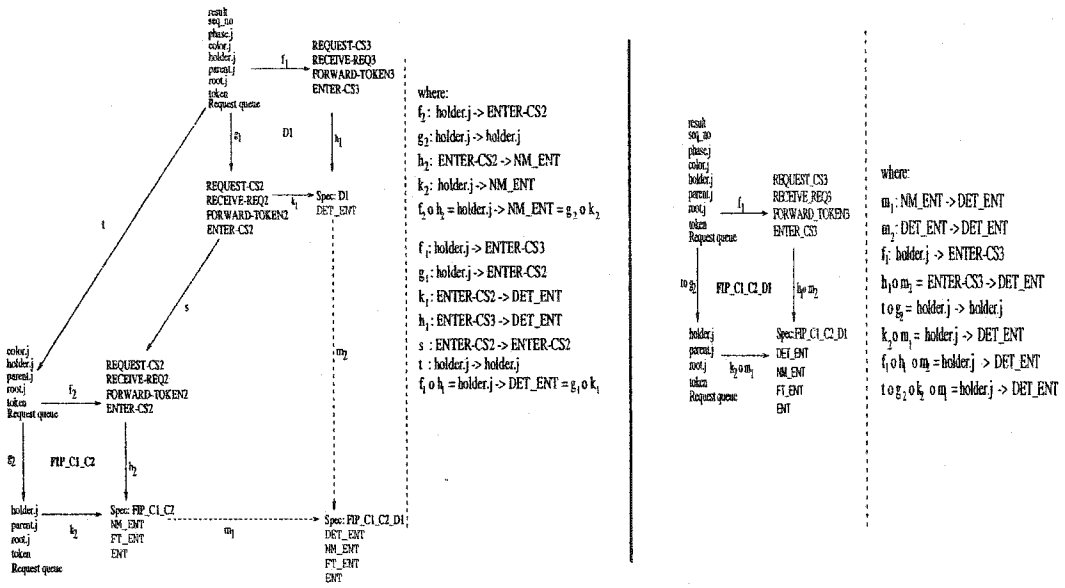

Figure 3.3: (a) Composition of *FIP_C1_C2* and Detector *D1* (b) Composed diagram of *FIP_C1_C2_D1*

section by having an additional feature that satisfies the safety condition $S1$. This enhanced property provided by $D1$ is termed as *DET_ENT*. The resulting specification (*FIP_C1_C2_D1*) has both the properties of *FIP_C1_C2* and *D1* since the morphism m_1 maps the *Spec:FIP_C1_C2* to *Spec:FIP_C1_C2_D1*, i.e., the property *NM_ENT* is present in the *Spec:FIP_C1_C2_D1*. Since the final specification *Spec:FIP_C1_C2_D1* has the property of *D1*, namely, *DET_ENT*, the morphism m_2 maps *DET_ENT* in *D1* to *DET_ENT* in the final specification. Figure 3.3(b) shows the composed diagram of *FIP_C1_C2* composed with *D1*. It has the properties of both *FIP_C1_C2* and *D1* for (a) accessing the critical section via *ENTER_CS2* and (b) detecting if any other process has the token before generating a new one, via operations explained in section 3.3.3. The composed module is thus able to detect and satisfy this safety condition before a new token is created. Another condition that should be satisfied

for the composite module *FIP_C1_C2_D1* to be converted to a *FTP* is that it should not execute the diffusing computation when forwarding the token. This condition is detected by composing *FIP_C1_C2_D1* with the detector *D2* to construct the overall *FTP* as shown in the next section.

3.4.4 Composing FIP_C1_C2_D1 and D2

Figure 3.4(a) shows the composition of the *FIP_C1_C2_D1* module and *D2*. The *D*2 module extends the features of *FIP_C*1*_C*2*_D*1 module by having an additional operation that satisfies the safety condition *S*2. The overall operation results in a masking access to enter the critical section. We call this property provided by *D*2 as *MASK_ENT*. The resulting specification (*FIP_C1_C2_D1_D2*) has both the properties of *FIP_C1_C2_D1* and *D2* since the morphism m_1 maps the *Spec:FIP_C1_C2_D1* to *Spec:FIP_C1_C2_D1_D2*, i.e., the *Spec:FIP_C1_C2_D1_D2* has the property *DET_ENT*. Since *Spec:FIP_C1_C2_D1_D2* has the property of *D2*, namely, *MASK_ENT*, the morphism m_2 maps *MASK_ENT* in *D2* to *MASK_ENT* in the final specification. Figure 3.4(b) shows the composed diagram of the *FIP_C1_C2_D1* composed with *D2*. It has the properties of both *FIP_C1_C2_D1* and *D2* for (a) entering the critical section via *ENTER_CS4* and (b) ensuring that the process is not participating in a diffusing computation when forwarding the token. This is done via modifications to operation *FORWARD_TOKEN3* resulting in the modified version *FORWARD_TOKEN4*. The composite module thus obtained is the final fault-tolerant

program.

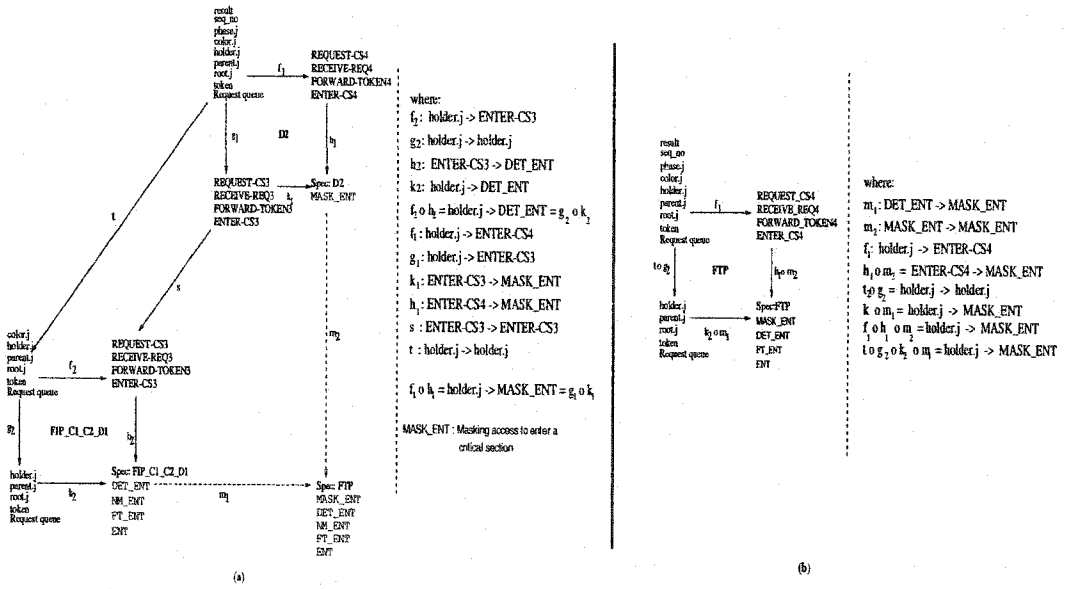

Figure 3.4: (a) Composition of *FIP_C1_C2_D1* and *D2* (b) Composed diagram of *FTP*

3.5 Compositional Specification and Verification Using Specware

In this section, we provide the specification and verification of the properties for the fault-tolerant mutual exclusion algorithm discussed in the previous section. We have used Kestrel's Specware tool [33] for the formal specification and verification of the case study. Specware[2] is a refinement-based approach to software development that supports *rigorous and explicit modularity* in the *specification and development* of software components. There is also a provision for refinement of an abstract specification,

[2]Details can be found at http://www.specware.org.

primarily done by refining algorithms and data structures. We emphasize that each step in the refinement process is constrained to preserve correctness. We describe the composition starting from the initial FIP and then composing the FIP with corrector and detector components resulting in the final composed program for mutual exclusion that is 'masking' fault-tolerant.

We initially specify the properties of the fault intolerant program associated with the following actions of:

- making a request to the holder process for obtaining the token (*Request_CS*).

  The property to be satisfied by a node for sending a request message to its holder node include:

  - the node should not have the privilege, i.e., , it should not be the holder.
  - the node wants the privilege for itself or for others (i.e., request queue is not empty),
  - a request message should not have been sent prior to this.

  This can be expressed formally as:

$$holder \neq self \wedge \neg asked \wedge (req\_q \neq empty \vee j \text{ needs to access } CS)$$

- transmitting a token to satisfy a pending request from a neighbour (*Fwd_Token*).

  The property to be satisfied by a node for sending a privilege message include:

  - the node should have the privilege, that is, it should be the holder.

- the node should not be using the critical section.
- the request queue of the node should not be empty, and
- the oldest request for the privilege must have come from another node.

This can be expressed in formal terms as:

$$holder = self \wedge \neg using \wedge req\_q \neq empty \wedge head(req\_q) \neq self$$

- receiving a request for the critical section from a neighbouring node (*Receive_Req*).
  - if the node is not the holder, it forwards the request to its holder if it has not been sent previously.
- receiving the token from a neighbouring node (*Enter_CS*).
  - if the node requested the token for itself, it enters CS, else it forwards the token to the process at the top of its request queue.

The FIP module imports *Address*, *RequestQ* and *CommonOperations* as its basic building blocks. The parameter of Address is formalized as sort using Specware and translated so that it can be used by other modules for their specification. The specification of *Address* module and the translation are specified below[3]:

```
Address = spec
  sort address = Nat
endspec

ADDRESS_to_ALL_TRANSLATION = translate(Address) by {address+->address}
```

[3]Specware code for FIP and FT components and corresponding proof can be found at the URL http://www.ece.concordia.ca/~sinha/specs/FTP.html.

The parameters of RequestQ are formalized as sorts and operations using Specware and translated so that it can be used by other modules for their specification, the only difference being the translation of this module consists of translations of the Address module too. The specification of *RequestQ* module is specified below:

```
RequestQ = spec
  import ADDRESS_to_ALL_TRANSLATION
  sort msg = Nat
  sort req_q = List address
  op empty : req_q -> Boolean
endspec

REQQ_to_ALL_TRANSLATION = translate(RequestQ) by { address +-> address,
req_q +-> req_q, empty +-> empty }
```

The CommonOperations specification provides the common functionality needed by all the properties of the mutual exclusion algorithm for sending and receiving of messages, enqueuing and dequeuing of messages.

```
CommonOperations = spec
  import REQQ_to_ALL_TRANSLATION
  op deq : req_q * address -> Boolean
  op dequeue : req_q -> address
  op enqueue : req_q * address -> Boolean
  op send : msg * address -> Boolean
  op receive : msg * address -> Boolean
  op self : address -> Boolean
endspec
```

The translation provided by this module is shown below:

```
COMOP_to_ALL_TRANSLATION = translate(CommonOperations) by
{address +-> address, req_q +-> req_q, empty +-> empty, deq +-> deq,
dequeue +-> dequeue, receive +-> receive, enqueue +-> enqueue,
send +-> send, self +-> self}
```

Having specified the parameters and operations needed for formalizing the distributed mutual exclusion protocol, we now specify various attributes (properties) of

the protocol in terms of its axioms. The properties are expressed as axioms, since we wish to extend these properties in a manner as to provide the same functionality, but with fault-tolerant features, with the addition of correctors and detectors.

The formal representation of the FIP specification in Specware is shown below. For clarity, only the sorts and axioms corresponding to the properties mentioned above are illustrated :

```
FIP = spec
  import COMOP_to_ALL_TRANSLATION
  sort using = Boolean
  sort holder = address
  sort myPid = Nat
  sort asked = Boolean
  sort parent = Nat
  ...
```

If the current process is the holder and it is not executing the critical section and there is a request in its queue, it dequeues the address in the queue and sets its holder to that address and forwards the token to that address.

```
  axiom assignPrivilege is fa(a:address,q:req_q,p:myPid,u:using,
                              ask:asked,h:holder,m:msg)
    isHolder(h,getSelf(p),p,u) & ~(usingCS(u,h,getSelf(p))) &
    ~(empty(q)) => updateHolder(dequeue(q),h) & unsetAsked(ask) &
    if (isHolder(h,getSelf(p),p,u)) then setUsing(u)
    else send(m,getAddress(h))
```

If the current process is not the holder and there is a request in its queue and it has not previously sent a REQUEST message, it sends the REQUEST message, adding it to its queue, and sets the *asked* sort to false.

```
axiom makeRequest is fa(h:holder,q:req_q,p:myPid,m:msg,a:asked,
                        u:using)
 ~(isHolder(h,getSelf(p),p,u) & ~(empty(q)) & ~(getAsked(a)) ) =>
 send(m,getAddress(h)) & setAsked(a)
 => receive(m,getAddress(h)) & addRequest(q,getSelf(p))
```

The *Request_CS* operation adds the process's request to its own request queue and if this is the privileged node, then *assignPrivilege* will allow this node to enter the critical section. If this is not the privilege node, *makeRequest* is invoked to send a REQUEST message to obtain the privilege.

```
axiom Request_CS is fa(h:holder,q:req_q,m:msg,p:myPid,a:address,
                       b:asked,u:using)
  if (isHolder(h,getSelf(p),p,u)) then
    addRequest(q,getSelf(p)) & assignPrivilege(a,q,p,u,b,h,m)
  else addRequest(q,getSelf(p)) & makeRequest(h,q,p,m,b,u)
```

If the process has finished executing the critical section and there is a request in its request queue, it sets *using* to false and initiates *assignPrivilege* which updates holder and subsequently forwards the TOKEN message to the process in the top of the queue.

```
axiom Fwd_Token is fa(a:address,q:req_q,u:using,m:msg,p:myPid,
                      h:holder,b:asked)
  (u = true) & (done executing CS) => (unsetUsing(u)  &
    assignPrivilege(a,q,p,u,b,h,m))

axiom Receive_Req is fa(x:address,q:req_q,m:msg,u:using,p:myPid,
                        ask:asked,h:holder)
  if( receive(m,x) & isHolder(h,getSelf(p),p,u) )
    then addRequest(q,x) & assignPrivilege(x,q,p,u,ask,h,m)
  else addRequest(q,x) & makeRequest(h,q,p,m,ask,u)

axiom Enter_CS is fa(a:address,q:req_q,m:msg,u:using,p:myPid,
                     ask:asked,h:holder)
  (receive(m,a) & updateHolder(getSelf(p),h) =>
    assignPrivilege(a,q,p,u,ask,h,m))
endspec
```

The translation provide by the FIP module which will be used by the *corrector* module for its operations is shown below.

```
FIP_to_ALL_TRANSLATION = translate(FIP) by { holder +-> holder,
Enter_CS +-> Enter_CS, Fwd_Token +-> Fwd_Token,
Receive_Req +-> Receive_Req, Receive_Token +-> Receive_Token }
```

3.5.1 Composing the FIP and C1 specifications

In the absence of any faults, the FIP with these actions results in the holder relationship being a directed tree rooted at the holder process. The addition of the correctors should ensure that the liveness property is satisfied. The liveness property to be satisfied is that, when an intermediate node fails, eventually the parent tree is reestablished and the holder relationship is also corrected in due course. Arora and Kulkarni propose two correctors for solving this problem, one for correcting the parent tree and the other for correcting the holder relation. This corrector also introduces new sorts for its functioning. The properties of $C1$ are given below:

$NT1 :: \forall(k : address)$

$colour(k) = green \wedge (Parent(k) \neq adj(k) \vee colour(Parent.k) = red) \Rightarrow$

$colour(k) := red.$

$NT2 :: \forall(j, k : address)$

$colour(k) = red \wedge (j \in adj(k) \wedge parent(j) \neq k) \Rightarrow$

$colour(k) := green; parent(k) := k; root(k) := k; holder(k) := k.$

$NT3 :: \forall(j, k : address)$

$(j \in adj(k) \wedge root(k) < root(j)) \wedge colour(k) = green \wedge colour(j) = green \Rightarrow$

$parent(k) := j; holder(k) := j; root(k) := j.$

The formal representation of the C1 specification in Specware is shown below. For clarity, all the sorts, operations and only the axioms corresponding to the properties

that are exported by this module are illustrated :

```
C1 = spec
  import FIP_to_ALL_TRANSLATION
  sort adjacent = List address
  sort children = List address
  sort pid = Nat
  sort colour = Boolean %% GREEN -> true , RED -> false
  op setRoot : address -> Boolean
  op getRoot : address -> address
  op setParent : address -> Boolean
  op getParent : address -> address
  op parent : address -> address
  op getAddr : Nat -> address
  op getStatus : address -> Boolean
  op getColour : address -> Boolean
  op setColour : colour * address -> Boolean
  ...
  ...
  axiom NT1 is  fa(c:colour,p:parent,a:adjacent)
   (c & (~(isProcessinList(a,getAddr(p))) or
   ~(getColour(getAddr(p))))) => setColour(~c,getAddress(p))

  axiom NT2 is fa(c:colour,k:address,adj:adjacent,q:req_q,
                  p:myPid,h:holder)
    (c & (fa (k) isProcessinList(adj,k) ) &
    ~(self(parent(k))) ) => setColour(c,getSelf(p)) &
    updateHolder(getSelf(p),h) & setParent(getSelf(p)) &
    setRoot(getSelf(p))

  axiom NT3 is fa(c:colour,k:address,h:holder,
                  adj:adjacent,p:myPid)
    (fa(k) isProcessinList(adj,k)) &
    (getRoot(getSelf(p)) < getRoot(k) & getColour(k) &
    getColour(getSelf(p))) =>
    setParent(k) & setRoot(getRoot(k)) & updateHolder(k,h)

  axiom Request_CS1 is fa(h:holder,q:req_q,m:msg,p:myPid,
                          a:address,b:asked,u:using,c:colour,
                          pa:parent,adj:adjacent)
    receive(m,a) & ~(send(m,a)) & Request_CS(h,q,m,p,a,b,u) =>
    NT1(c,pa,adj) & NT2(c,a,adj,p,h) & NT3(a,h,adj,p) =>
    (~(getStatus(a)) & (a = getParent(getSelf(p))))

  axiom Fwd_Token1 is fa(a:address,q:req_q,u:using,m:msg,
                         p:myPid,h:holder,b:asked,pa:parent,
                         adj:adjacent,c:colour)
    receive(m,a) & ~(send(m,a)) & Fwd_Token(a,q,u,m,p,h,b) =>
    NT1(c,pa,adj) & NT2(c,a,adj,p,h) & NT3(a,h,adj,p) =>
   (~(getStatus(a)) & (a = getParent(getSelf(p))))
```

```
  axiom Receive_Req1 is f(a:address,q:req_q,m:msg,u:using,
                          p:myPid,ask:asked,h:holder,c:colour,
                          pa:parent,adj:adjacent)
    receive(m,a) & ~(send(m,a)) & Receive_Req(a,q,m,u,p,ask,h) =>
    NT1(c,pa,adj) & NT2(c,a,adj,p,h) & NT3(a,h,adj,p) =>
    (~(getStatus(a)) & (a = getParent(getSelf(p))))

  axiom Enter_CS1 is fa(a:address,q:req_q,m:msg,u:using,
                        p:myPid,ask:asked,h:holder,c:colour,
                        pa:parent,adj:adjacent)
    receive(m,a) & ~(send(m,a)) & Enter_CS(a,q,m,u,p,ask,h) =>
    NT1(c,pa,adj) & NT2(c,a,adj,p,h) & NT3(a,h,adj,p) =>
    (~(getStatus(a)) & (a = getParent(getSelf(p))))
endspec
```

The translation is done the same way as before, but this time it would have what FIP module had translated, and also the sorts, operations and properties of the C1 module.

```
FIPC1_to_ALL_TRANSLATION = translate(C1) by { holder +-> holder,
Request_CS1 +-> Request_CS1, Fwd_Token1 +-> Fwd_Token1,
Receive_Req1 +-> Receive_Req1, Enter_CS1 +-> Enter_CS1 }
```

In order to compose the FIP and C1 modules, we have to specify the various morphisms that link them. We formalize the morphisms between these two specifications in the following manner:

```
FIP_TO_C1_MORPHISM = morphism FIPgm -> C1 { holder +-> holder,
Request_CS +-> Request_CS1, Fwd_Token +-> Fwd_Token1,
Receive_Req +-> Receive_Req1, Enter_CS +-> Enter_CS1 }
```

We then define the diagram with FIP and C1 specifications as the nodes, and the morphism as the link between them.

```
FIP_C1 = diagram {a +-> FIPgm, b +-> C1, i: a->b +-> morphism
FIPgm -> C1 {holder+->holder, Request_CS +-> Request_CS1,
Fwd_Token +-> Fwd_Token1, Receive_Req +-> Receive_Req1,
Enter_CS +-> Enter_CS1  }}
```

We finally construct the composite specification of the FIP and C1 modules by taking the colimit of the diagram as shown below:

```
FIPC1 = colimit FIP_C1
```

3.5.2 Composing the FIP_C1 and C2 specifications

We formalize the Corrector C2 by first importing what the previous tranlation offers, and adding the parameters needed for defining C2's properties. The actions of the second corrector are given below:

$$NH1 :: \forall(j : address)$$

$$holder(j) \neq j \wedge holder(j) \neq parent(j) \wedge (holder(j) \notin adj(j)) \vee$$

$$parent(j) \neq j \wedge holder(j) \neq parent(j) \wedge holder(parent(j)) \neq j \Rightarrow holder(j) := j$$

$$NH2 :: \forall(j : address, k : address)$$

$$holder(j) = k \wedge holder(k) = j \wedge j \neq k \wedge parent(j) = k \Rightarrow holder(j) = parent(j)$$

The formal representation of the C2 specification in Specware is shown below. For clarity, all the sorts, operations and only the axioms corresponding to the properties that are exported by this module are illustrated :

```
C2 = spec
  import FIPC1_to_ALL_TRANSLATION
  ...
  ...
  axiom NH1 is fa(h:holder,pid:myPid,c:child_q)
    (( ~(h = getSelf(pid)) & ~(h = getParent(getSelf(pid))) &
    ~(member(h,c)) ) or (~(getParent(getSelf(pid)) = getSelf(pid)) &
    ~(h = getParent(getSelf(pid))) &
    ~(getHolder(getParent(getSelf(pid)))=getSelf(pid)) ) ) =>
    updateHolder(getParent(getSelf(pid)),h)
```

```
  axiom NH2 is fa(h:holder,k:address,p:myPid)
    ((h = k) & ( getSelf(p) = getHolder(k) ) & ~(k = getSelf(p)) &
    (k = getParent(p)) ) => updateHolder(getParent(getSelf(p)),h)

(* holder=self & Request_CS1 => initiate NH1 & NH2 *)

  axiom Request_CS2 is fa(h:holder,q:req_q,m:msg,p:myPid,
  a:address,b:asked,u:using,c:colour,
                          pa:parent,adj:adjacent,cq:child_q)
    h = getSelf(p) & Request_CS1(h,q,m,p,a,b,u,c,pa,adj) =>
    NH1(h,p,cq) & NH2(h,a,p)

(* holder=self & Fwd_Token1 => initiate NH1 & NH2 *)

  axiom Fwd_Token2 is fa(a:address,q:req_q,u:using,m:msg,
                         p:myPid,h:holder,b:asked,pa:parent,
                         adj:adjacent,c:colour,cq:child_q)
    h=getSelf(p) & Fwd_Token1(a,q,u,m,p,h,b,pa,adj,c) =>
    NH1(h,p,cq) & NH2(h,a,p)

  axiom Receive_Req2 is fa(a:address,q:req_q,m:msg,u:using,
                           p:myPid,ask:asked,h:holder,c:colour,
                           pa:parent,adj:adjacent,cq:child_q)
    h = getSelf(p) & Receive_Req1(a,q,m,u,p,ask,h,c,pa,adj) =>
    NH1(h,p,cq) & NH2(h,a,p)

  axiom Enter_CS2 is fa(a:address,q:req_q,m:msg,u:using,
                        p:myPid,ask:asked,h:holder,c:colour,
                        pa:parent,adj:adjacent,cq:child_q)
    h = getSelf(p) & Enter_CS1(a,q,m,u,p,ask,h,c,pa,adj) =>
    NH1(h,p,cq) & NH2(h,a,p)
endspec
```

The translation is done the same way as before, but this time it would have what C1 module had translated, and also the sorts, operations and properties of the C2 module.

```
FIPC1C2_to_ALL_TRANSLATION = translate(C2) by { holder +-> holder,
Request_CS2 +-> Request_CS2 , Fwd_Token2 +-> Fwd_Token2,
Receive_Req2 +-> Receive_Req2, Enter_CS2 +-> Enter_CS2 }
```

In order to compose the C1 and C2 modules, we have to specify the various morphisms that link them. We formalize the morphisms between these two specifications

in the following manner:

```
FIP_C1_TO_C2_MORPHISM = morphism C1 -> C2 { holder +-> holder,
Request_CS1 +-> Request_CS2, Fwd_Token1 +-> Fwd_Token2,
Receive_Req1 +-> Receive_Req2, Enter_CS1 +-> Enter_CS2 }
```

We then define the diagram with C1 and C2 specifications as the nodes, and the morphism as the link between them.

```
FIP_C1_C2 = diagram {a +-> C1,b +-> C2,i: a->b +-> morphism
C1 -> C2 { holder +-> holder, Request_CS1 +-> Request_CS2,
Fwd_Token1 +-> Fwd_Token2, Receive_Req1 +-> Receive_Req2,
Enter_CS1 +-> Enter_CS2 }}
```

We finally construct the composite specification of the C1 and C2 modules by taking the colimit of the diagram as shown below:

```
FIPC1C2 = colimit FIP_C1_C2
```

3.5.3 Composing the FIP_C1_C2 and D1 specifications

The corrector actions ensure that the parent and holder relationships are set right. Once this is achieved, the detector actions come into play to ensure that there is a unique token being circulated among the nodes. The actions of the first detector are given below:

$$Init :: \forall(j : address)$$

$$parent(j) = j \Rightarrow seq(j) := seq(j) + 1; result := true; phase(j) := false$$

$$Prop :: \forall(j : address) root(j) = root(parent(j)) \wedge$$

$$seq(j) = seq(parent(j)) \Rightarrow seq(j) := seq(parent(j)) \wedge$$

$if\ (phase(parent(j)) \wedge holder(j) = parent(j))\ then$

$phase(j) := true; result := true$

$else\ result := false$

$Abort :: \forall(j : address) phase(j) := true \wedge phase(j) := false \wedge$

$(parent(j) \in adj(j)) \Rightarrow send(m, parent(j))$

$Complete :: \forall(j, k : address)$

$phase(j) \wedge (k \in adj(j) \wedge (root(j) = root(k) \wedge seq(j) = seq(k))$

$\wedge(k \in child(j) \wedge \neg phase(k)) \Rightarrow result(j) := result(k); phase(j) := true;$

$if(parent(j) = j \wedge \neg result(j))\ then\ Init(j)$

$else\ result(j) := false$

The formal representation of the D1 specification in Specware is shown below. For clarity, all the sorts, operations and only the axioms corresponding to the properties that are exported by this module are illustrated :

```
D1 = spec
  import FIPC1C2_to_ALL_TRANSLATION
  ...
  ...
  axiom Init is fa(p:myPid,ph:phase,s:seq,r:result)
    (getSelf(p)=getParent(getSelf(p))) => (newseq(s) &
    (r = true) & (ph = false))

  axiom Prop is fa(p:myPid,ph:phase,s:seq,r:result,h:holder)
    ((getRoot(getSelf(p))=getRoot(getParent(getSelf(p)))) &
    (s=getSeq(getParent(getSelf(p))))) =>
    setSeq(getSelf(p),getSeq(getParent(getSelf(p)))) &
    (if(getPhase(getParent(getSelf(p))) &
        (h=getParent(getSelf(p))))
      then ((ph = true)  & (r = true))
    else (r = false))
```

```
axiom Abort is fa(ph:phase,r:result,m:msg,
  adj:adjacent,p:parent)
  setPhase(ph) & (r = false) &
  isProcessinList(adj,getAddress(p)) => send(m,getAddress(p))


axiom Complete is fa(ph:phase,r:result,s:seq,p:parent,
                     adj:adjacent,k:address,ch:child_q,pid:myPid)
(getPhase(getSelf(pid)) & isProcessinList(adj,k) &
(getRoot(getSelf(pid))=getRoot(k)) & (getSeq(getSelf(pid))=getSeq(k))
& (member(k,ch)) & ~(getPhase(getSelf(pid)))) =>
(r = getResult(k)) & setPhase(ph) &
(if (( getParent(getSelf(pid))=getSelf(pid)) & (r = false))
     then Init(pid,ph,s,r)
 else (r = false))
```

If *Request_CS2* results in *holder* and *root* to be *self*, then diffusion algorithm is initiated since this process is the root, and if the algorithm is not aborted, it implies that the diffusion completed successfully.

```
axiom Request_CS3 is fa(h:holder,q:req_q,m:msg,p:myPid,a:address,
                        b:asked,u:using,c:colour, pa:parent,
                        adj:adjacent,cq:child_q,ph:phase,
                        r:result,s:seq)
  if ( h=getSelf(p) & getRoot(getSelf(p))=getSelf(p) ) then
    Request_CS2(h,q,m,p,a,b,u,c,pa,adj,cq) & Init(p,ph,s,r) &
    ~(Abort(ph,r,m,adj,pa)) => Complete(ph,r,s,pa,adj,a,cq,p)
  else
    Request_CS2(h,q,m,p,a,b,u,c,pa,adj,cq) & Prop(p,ph,s,r,h) &
    ~(Abort(ph,r,m,adj,pa)) => Complete(ph,r,s,pa,adj,a,cq,p)

axiom Fwd_Token3 is fa(a:address,q:req_q,u:using,m:msg,p:myPid,
                       h:holder, b:asked,pa:parent,adj:adjacent,
                       c:colour,cq:child_q, ph:phase,r:result,s:seq)
  if ( h=getSelf(p) & getRoot(getSelf(p))=getSelf(p) ) then
    Fwd_Token2(a,q,u,m,p,h,b,pa,adj,c,cq) & Init(p,ph,s,r) &
    ~(Abort(ph,r,m,adj,pa)) => Complete(ph,r,s,pa,adj,a,cq,p)
  else
    Fwd_Token2(a,q,u,m,p,h,b,pa,adj,c,cq) & Prop(p,ph,s,r,h) &
    ~(Abort(ph,r,m,adj,pa)) => Complete(ph,r,s,pa,adj,a,cq,p)

axiom Receive_Req3 is fa(a:address,q:req_q,m:msg,u:using,p:myPid,
                         ask:asked,h:holder,c:colour,pa:parent,
                         adj:adjacent,cq:child_q, ph:phase,
                         r:result,s:seq)
if ( h=getSelf(p) & getRoot(getSelf(p))=getSelf(p) ) then
  Receive_Req2(a,q,m,u,p,ask,h,c,pa,adj,cq) & Init(p,ph,s,r) &
```

```
    ~(Abort(ph,r,m,adj,pa)) => Complete(ph,r,s,pa,adj,a,cq,p)
  else
    Receive_Req2(a,q,m,u,p,ask,h,c,pa,adj,cq) & Prop(p,ph,s,r,h) &
    ~(Abort(ph,r,m,adj,pa)) => Complete(ph,r,s,pa,adj,a,cq,p)

  axiom Enter_CS3 is fa(a:address,q:req_q,m:msg,u:using,p:myPid,
                        ask:asked,h:holder,c:colour,pa:parent,
                        adj:adjacent,cq:child_q,ph:phase,
                        r:result,s:seq)
    if ( h=getSelf(p) & getRoot(getSelf(p))=getSelf(p) ) then
      Enter_CS2(a,q,m,u,p,ask,h,c,pa,adj,cq) & Init(p,ph,s,r) &
      ~(Abort(ph,r,m,adj,pa)) => Complete(ph,r,s,pa,adj,a,cq,p)
    else
      Enter_CS2(a,q,m,u,p,ask,h,c,pa,adj,cq) & Prop(p,ph,s,r,h) &
      ~(Abort(ph,r,m,adj,pa)) => Complete(ph,r,s,pa,adj,a,cq,p)
endspec
```

The translation of the D1 module is shown below.

```
FIPC1C2D1_to_ALL_TRANSLATION = translate(D1) by {holder +-> holder,
Request_CS3 +-> Request_CS3, Fwd_Token3 +-> Fwd_Token3,
Receive_Req3 +-> Receive_Req3, Enter_CS3 +-> Enter_CS3}
```

In order to compose the C2 and D1 modules, we have to specify the various morphisms that link them. We formalize the morphisms between these two specifications in the following manner:

```
FIP_C1_TO_C2_TO_D1_MORPHISM = morphism C2 -> D1 {holder +-> holder,
Request_CS2 +-> Request_CS3, Fwd_Token2 +-> Fwd_Token3,
Receive_Req2 +-> Receive_Req3, Enter_CS2 +-> Enter_CS3}
```

We then define the diagram with C2 and D1 specifications as the nodes, and the morphism as the link between them.

```
FIP_C1_C2_D1 = diagram {a +-> C2,b +-> D1,i: a->b +-> morphism
C2 -> D1 {holder +-> holder, Request_CS2 +-> Request_CS3,
Fwd_Token2 +-> Fwd_Token3, Receive_Req2 +-> Receive_Req3,
Enter_CS2 +-> Enter_CS3}}
```

We finally construct the composite specification of the C2 and D1 modules by taking the colimit of the diagram as shown below:

```
FIPC1C2D1 = colimit FIP_C1_C2_D1
```

3.5.4 Composing the FIP_C1_C2_D1 and D2 specifications

The final module to be composed to obtain the masking fault-tolerant mutual exclusion algorithm involves the addition of the detector D2. The detector action involves addition of a predicate to ensure the *phase* sort is enabled. The formal representation of the D2 specification in Specware is shown below.

```
D2 = spec
  import FIPC1C2D1_to_ALL_TRANSLATION
  ...
  ...
  theorem Request_CS4 is fa(h:holder,q:req_q,m:msg,p:myPid,
                            a:address, b:asked,u:using,c:colour,
                            pa:parent,adj:adjacent, cq:child_q,ph:phase,
                            r:result,s:seq)
    if ((Request_CS3(h,q,m,p,a,b,u,c,pa,adj,cq,ph,r,s) & (ph = true)))
      then ( h = getSelf(p))
    else ~( h = getSelf(p))

  theorem Fwd_Token4 is fa(a:address,q:req_q,u:using,m:msg,p:myPid,
                           h:holder,b:asked,pa:parent,adj:adjacent,
                           c:colour, cq:child_q,ph:phase,r:result,s:seq)
    if (Fwd_Token3(a,q,u,m,p,h,b,pa,adj,c,cq,ph,r,s) & (ph = true)) then
      deq(q,getAddress(h))
    else ~(deq(q,getAddress(h)))

  theorem Enter_CS4 is fa(a:address,q:req_q,m:msg,u:using,p:myPid,
                          ask:asked,h:holder,c:colour,pa:parent,
                          adj:adjacent,cq:child_q,ph:phase,r:result,s:seq)
    if (( Enter_CS3(a,q,m,u,p,ask,h,c,pa,adj,cq,ph,r,s) & (ph = true)))
      then (h = getSelf(p))
    else ~( h = getSelf(p))

  theorem Receive_Req4 is fa(a:address,q:req_q,m:msg,u:using,p:myPid,
                             ask:asked,h:holder,c:colour,pa:parent,
                             adj:adjacent,cq:child_q,ph:phase,r:result,s:seq)
    if ((Receive_Req3(a,q,m,u,p,ask,h,c,pa,adj,cq,ph,r,s) & (ph = true)))
      then (h = getSelf(p))
    else ~(h = getSelf(p))
endspec
```

The translation of the D2 module, which is the final export and can be used by any other module whose requirement is to have a fault-tolerant mutual exclusion, is

shown below.

```
FIPC1C2D1D2_to_ALL_TRANSLATION = translate(D2) by {holder +-> holder,
Request_CS4 +-> Request_CS4, Fwd_Token4 +-> Fwd_Token4,
Receive_Req4 +-> Receive_Req4, Enter_CS4 +-> Enter_CS4}
```

In order to compose the D1 and D2 modules, we have to specify the various morphisms that link them. We formalize the morphisms between these two specifications in the following manner:

```
FIP_C1_TO_C2_TO_D1_D2_MORPHISM = morphism D1 -> D2 {holder +-> holder,
Request_CS3 +-> Request_CS4, Fwd_Token3 +-> Fwd_Token4,
Receive_Req3 +-> Receive_Req4, Enter_CS3 +-> Enter_CS4}
```

We then define the diagram with D1 and D2 specifications as the nodes, and the morphism as the link between them.

```
FIP_C1_C2_D1_D2 = diagram {a +-> D1, b +-> D2, i: a->b +-> morphism
D1 -> D2 {holder +-> holder, Request_CS3 +-> Request_CS4,
Fwd_Token3 +-> Fwd_Token4, Receive_Req3 +-> Receive_Req4,
Enter_CS3 +-> Enter_CS4}}
```

We finally construct the composite specification of the D1 and D2 modules by taking the colimit of the diagram as shown below:

```
FIPC1C2D1D2 = colimit FIP_C1_C2_D1_D2
```

Utilizing the various axioms provided by the individual specifications of *FIP, C1, C2, D1 and* D2, we formulate the theorem *Enter_CS4*, *Receive_Req4*, *Request_CS4* and *Fwd_Token4* (as shown above), required for proving the final property of *Entering the CS*, *Receiving a request*, Requesting a CS and *Forwarding the token.* We finally verify the above mentioned property by processing the above specification along with the theorem in Specware with a built-in interface to Snark theorem prover. The statements for the proof of these two properties are as given below:

```
p1 = prove Enter_CS4 in D2 using Enter_CS3 send receive
p2 = prove Receive_Req4 in D2 using Receive_Req3 send receive
p3 = prove Request_CS4 in D2 using Request_CS3 send receive
p4 = prove Fwd_Token4 in D2 using Fwd_Token3 send receive
```

In this Chapter, we have shown the feasibility of our approach through a case study of the Mutual Exclusion algorithm. In the next Chapter, we illustrate an additional case study and show the design of the detector and corrector components for a multimedia protocol.

Chapter 4

Case Study 2: The Label Distribution Protocol

In this chapter, we illustrate the application of our concepts with a case study of the Label Distribution Protocol (LDP) [1] which is a part of the Multi Protocol Label Switching (MPLS) [29] architecture. This Chapter is organized as follows: Section 4.1 discusses the need for fault-tolerance in the LDP protocol. Section 4.2 describes the case study and gives a brief description of the FIP, corrector and detector modules. Section 4.3 presents the proof of the design of the fault-tolerant components. Section 4.4 details our approach of composing the FIP, corrector and detector for the case study. Section 4.5 presents the formal specification and verification of the fault-tolerant version of LDP using MetaSlang language of Specware.

4.1 Need for fault-tolerance in LDP

The significant presence and coverage of the Internet has led to an increasing number of new real-time interactive services that are being offered or are still in their infancy. A majority of these services involves communication between one or more computer systems. Current communication protocols are built on existing transport layer protocols that provide both reliable and unreliable service for exchanging information. Some of the faults that may occur in such a scenario are the failure of the transport protocol stack. Recovery in such cases is worsened by the fact that some control messages may have been lost during the failure, leading to inconsistency in the state information at the communicating nodes.

MPLS is a fast switching technology in which the packets are forwarded based on labels assigned to the packets. Since the forwarding is done based on labels, a protocol is needed for exchanging this label information between the label switched routers (LSRs). RFC3036 defines a set of procedures called the LDP for exchanging label information by which LSRs distribute labels to support MPLS forwarding. LDP uses TCP as the underlying protocol for communication with the neighbouring LSRs. A failure in the TCP protocol stack could lead to a connection failure between the communicating LSRs. RFC3479 proposes enhancements to the LDP to achieve fault-tolerance. In our case study, we show that the final FT-LDP program is masking tolerant because, in the presence of a communication fault, it still satisfies its specification, by reopening a new TCP connection and recovering to a state from

where normal execution is restored.

4.2 Identifying the Fault-Intolerant Program and Fault-Tolerant Components

Adding masking fault-tolerance to a FIP involves two tasks: (1) of ensuring that the program recovers to states from where it satisfies both the safety and liveness specification, and (2) of ensuring that the program does not violate the safety specification during recovery. Below, we show that the fault-tolerant LDP can be designed by first designing a fault-intolerant program, LDP, and then composing it with detector components, followed by correctors. The subsequent sections propose a component-based approach to adding fault-tolerant enhancements to LDP as suggested in [15].

4.2.1 Identifying the Fault Intolerant Program

The working of the Label Distribution Protocol[1] without any fault-tolerant features is explained below:

The Multi-Protocol Label Switching (MPLS) architecture proposes integrating Layer 3 routing and Layer 2 switching functionalities [29] and is rapidly becoming a key technology for use in core networks. It advocates the use of a short four byte label to introduce connection-oriented features into the Internet Protocol (IP) by replacing

[1]We shall call this module as LDP.

the routing of IP packets based on information in IP header with the light-weight label. The approach presented in [1] defines a set of procedures for exchanging label information by which LSRs distribute labels to support MPLS forwarding. A brief definition of some of the terms needed to comprehend the protocol are given below:

- **Forward Equivalence Class (FEC):** a group of packets having the same forwarding treatment.
- **Label:** a short fixed length identifier used to identify a FEC.
- **Label Switched Path:** the path through one or more LSRs followed by packets in a particular FEC.
- **LDP peers:** any two LSRs which communicate with one another using LDP for the exchange of label/FEC mapping are referred to as LDP peers with respect to that mapping.

Some of the assumptions made for the working of this protocol are: A reliable and ordered delivery of messages is assumed for the label distribution operation. TCP is used to provide this service.

We explain only the basic operations of the protocol and the actions of the protocol that are affected by faults. The operation of LDP can be described as consisting of the following steps:

1. the initial discovery phase in which a LSR discovers potential LDP peers.
2. the session establishment and maintenance phase in which the LDP session is set

up between the peers, and label and address messages are exchanged between the peers after successful setup, and,

3. finally session closure when the label space is no longer required.

Discovery Mechanism:

This process enables an LSR to discover its potential peers that can communicate using LDP thereby avoiding the static configurations of an LSR's peers. There are two variants of this mechanism, the first one called the basic discovery mechanism involves periodic multicasting of the *Hello* message to the "all routers on this subnet" multicast address, thereby locating LSR neighbours that are directly connected at the link level, the second involves sending a targeted *Hello* message and is called the extended discovery mechanism to locate LSR neighbours not directly connected at the link level.

Reception of a *Hello* message identifies a hello-adjacency with a potential peer that can be reached on the interface, and the label space it intends to use for the interface in the case of basic discovery mechanism. In the case of extended discovery, the received *Hello* identifies a hello-adjacency with a potential peer reachable at network level and the label space it wishes to use. The LSR receiving the targeted *Hello* may choose to respond, and if it does so, it sends periodic targeted *Hello* messages to the sender.

Session Establishment and Maintenance

Establishment of a hello-adjacency initiates the session establishment phase, which is a two-step process. The first involves a transport connection establishment and the second is session initialization.

Each LSR determines its peer LSR's transport address either through the *Address* Type, Length, Value (TLV) present in the *Hello* message or the source address of the *Hello* message. If there is no session setup for the exchange of the label spaces in the *Hello* message, a TCP connection is established with the LSR, whose address is greater than the peer address, acting as the active LSR. The passive LSR waits for the active LSR to establish the connection. After a TCP connection is established between the two LSRs, they negotiate session parameters using the *Initialization* message. Successful initialization results in a LDP session between the two LSRs. The state machine diagram for the protocol operation is shown in [15]. The session is setup successfully once the LSR's reach *Operational* state and label messages can then be exchanged between the two peers.

Session Closure

The request for a label or advertising a label to a peer is a local decision done by an LSR, i.e., it requests for a label mapping when it needs one, or advertises a label mapping when it wants the neighbour to use the label. When the LSR does not wish to use the label space anymore it closes the LDP session by sending a *Shutdown*

message and closing the TCP connection after withdrawing the label. The basic sorts of the protocol are:

- **ldpid:** an LDP identifier identifies a LSR's label space and is a combination of the LSR identifier and the label space the LSR wishes to advertise. An LSR that advertises multiple label spaces uses different LDP identifiers for each label space.
- **peer:** the address of the peer LSR.
- **self:** the address of this LSR.
- **mode:** active/passive LSR.
- **Hello_msg:** the sort representing the *Hello* message.
- **Init_msg:** the sort representing the *Initialization* message.
- **Label_msg:** the sort representing the *Label* message.
- **Notification_msg:** the sort representing the *Notification* message.

The following are the basic operations of the protocol:-

- **SEND_HELLO:** send *Hello* message to all neighbours in case of basic discovery, or else send the *Hello* message to a specific router in case of extended discovery.
- **RECEIVE_HELLO:** this operation is invoked on reception of the *Hello* message.
- **send:** this is the basic operation used to send any LDP message.

- **receive:** this is the basic operation used to receive any LDP message.
- **RX_KA_TO:** this function is called on the expiry of the KeepAlive timer.
- **ldp_Init:** initializes the LDP session.
- **ldp_state_mc:** the function that handles LDP state machine.
- **close:** closes the LDP session. All state information is discarded and the label cross connects are discarded.

4.2.2 Identifying the Faulty Scenarios

MPLS is a technology that is to be deployed in core routers and as such, these systems must have a low downtime. Thus these systems have to exploit the fault tolerant hardware/software to provide high availability. LDP being one of the components of MPLS, [15] identifies some of the issues (explained below) that make it difficult to implement a fault tolerant LSR using the current LDP protocol, and has defined fault-tolerant enhancements to the LDP specification.

Since TCP is used for reliable data transfer as explained in Section 4.2.1, a failure of the TCP or LDP protocol stack will result in a connection failure between the peer LSR's.

Any detection and recovery mechanisms to detect these faults need changes only to the control plane without disrupting the data plane. This is because many of the current router architectures isolate the control plane from the data plane, and hence traffic in the data plane can persist during the recovery of the control plane.

Providing a fault-tolerant implementation of TCP would involve saving all sent and received messages, which is a tough proposition. Recovery from TCP connection failure because of failure in TCP or LDP stacks is exacerbated by the fact that LDP control messages may have been lost during the connection failure leading to inconsistency in the state information at the LDP peers.

The assumptions we make for describing the fault tolerant components are:

- the LDP peers wish to use fault-tolerant (FT) enhancements and restore their state on reconnection in case of a TCP connection failure.
- both the peers wait for the duration of the FT Reconnection timeout before releasing state information, i.e., they set the "FT Reconnect" flag to *one* after reconnection, and,
- the TCP connection is restored after a failure.

Identifying the Detector Components

The detector component ensures that the program does not deviate from its normal execution and that any bad state is detected by this component. Observe that *LDP* violates its safety specification from states where there is a connection failure between the peers. Thus the safety condition to be identified is the failure of the TCP connection. We construct detectors that ensure that the safety specification is preserved. A few ways in which the TCP connection failure can be detected[2] are:-

[2]We consider only the fourth and the sixth approach for detection.

- indication from the management component that a TCP connection or underlying resource is no longer active.
- notification from a hardware management component of an interface failure.
- sockets keepalive timeout.
- sockets send failure.
- new (incoming) Socket opened.
- LDP protocol timeout.

Detector D1

The first detector throws an exception when the *send* operation fails due to a TCP connection failure. When the TCP connection fails, the socket *send* operation that invokes the TCP/IP protocol stack fails. This exception condition is used to set the boolean sort $tcp\_cx\_failure$ which indicates the TCP connection failure.

Detector D2

This detector is the watchdog timer function that is modified to indicate a TCP connection failure. This exception also sets the boolean sort $tcp\_cx\_failure$ which is used to indicate the TCP connection failure[3]. Furthermore a FT Protection TLV is added to all messages thereby modifying those sorts that represent the LDP messages.

[3]In the case of the FIP, the session is closed on expiry of this timer.

Identifying the Corrector Components

The liveness property to be satisfied is that, on restoration of the TCP connection, the protocol operation is restored to a state from where normal operation can continue. On detection of the TCP connection failure, a "FT Reconnection" timer is started and the "hold" timer is stopped for the duration of the Reconnection timer. The active LSR then initiates the recovery operation. RFC3479 proposes two ways to achieve this, corrector C1, based on handshaking mechanism and C2, based on checkpointing. The timer actions are common to both the correctors.

Corrector C1

Handshaking of the LDP messages is achieved through the use of acknowledgments (ACKs). The original message and the ACK are co-related using a sequence number carried in the FT Protection TLV and returned in the ACK. After restoration of the TCP connection between the peers, any sequence numbered LDP messages that were lost, are re-issued. On reception of a re-issued message, an LSR processes it as if received for the first time. Any operations that were pending during the connection failure must also be transmitted on restoration of the TCP connection. The initialization messages exchanged on reinitialization of the session include the ACKs for the last message each peer received and logged before the failure of the TCP connection. Based on these, the LSRs determine the sequence number from

which messages have to be re-issued[4].

This corrector adds the following sorts:

- **seq:** sequence number of the message to be sent next by the sender.
- **ack_seq_no:** sequence number of the last message received by the receiver.
- **rs:** boolean sort set to *true* to indicate if the sender received an ACK for the last message it sent.
- **rr:** boolean sort set to *true* to indicate if the receiver received a message, but has not yet sent an ACK.
- **msg_q:** the 0 or 1 messages/ACKs in transit are indicated by this queue, we have two instances of this queue to indicate the transmit and receive channels.

The operations of $C1$ are given below:

- **sendLog:** this operation modifies the *mod_send* operation and logs any message after sending to the peer LSR.
- **startRecxTimer:** this operation starts the FT Reconnection timer on detection of connection failure.
- **reset_rx_hold_to:** this operation stops the Hold timer. It is invoked when the Reconnection timer is initiated.
- **ReInitialize:** this operation reinitializes the LDP session. If this LSR is active LSR it initiates a new TCP connection to restore the LDP session. The sequence

[4]We assume that the sender waits for an acknowledgment before sending the next message. For ease of construction of the corrector, we are not considering the accumulation of Acks, which is allowed by the FT-LDP as explained in [15]

numbers from where the LSRs have not received ACKs are exchanged during this operation.

- **Reissue:** this operation reissues all the messages on recovery of TCP connection, starting from the sequence number exchanged in the *ReInit* operation.
- **send_msg:** this operation sends a sequence numbered message to the receiver.
- **rx_ack:** this operation receives an acknowledgement from the receiver.
- **rx_msg:** this operation receives a message sent from the sender.
- **send_ack:** this operation sends an acknowledgement to the sender.

Corrector C2

Checkpointing is an operation that involves periodic logging of the program state so that when a failure occurs, a program can be restored to a previous state based on the logged messages. Checkpointing in LDP is achieved by using the *KeepAlive* messages as checkpointing messages. In this method, when the C bit [15] in the FT Protection TLV is set, it is an indication to use the checkpointing procedures. "Check-Pointing" in the context of the fault tolerant LDP refers to the process of message exchanges that confirm receipt and processing (or secure storage) of specific protocol messages [15]. On restoring the TCP connection, any LDP messages that were lost, are re-issued. This is achieved by the recovery algorithm, which ensures that any lost messages are resent. On reception of a re-issued message, an LSR processes it as if received for the first time. Any operations that were pending during the connection failure

must also be transmitted on restoration of the TCP connection. The initialization messages exchanged on reinitialization of the session include the sequence numbers of the checkpointing message from where recovery has to be initiated.

This corrector adds the following sorts:

- **kaseq_no:** sequence number for the *KeepAlive* message.
- **last_kaseq_no:** last *KeepAlive* sequence number successfully acknowledged to the peer.

The operations of $C2$ are given below:

- **sendLog:** this operation modifies the *mod_send* operation and logs any message after sending to the peer LSR.
- **startRecxTimer:** this operation starts the FT Reconnection timer on detection of connection failure.
- **reset_rx_hold_to:** this operation stops the Hold timer. It is invoked when the Reconnection timer is initiated.
- **ft_ldp_state_mc:** this operation modifies the *ldp_state_mc* to flush the state on the reception of a KeepAlive message.
- **ft_KA_timer:** this operation modifies the *mod_KA_timer* to initiate checkpointing procedures.
- **ReInit:** this operation reinitializes the LDP session. If this LSR is active LSR it initiates a new TCP connection to restore the LDP session. This operation involves the exchange of the sequence numbers of the secured check-points.

- **Recover:** this operation initiates the recovery procedure to ensure that there are no lost messages during the duration of the TCP connection failure.

So far, we have discussed the FIP, and the design of the detector and corrector components for the FIP. Next we show the proof of correctness of the detector and correctors presented above.

4.3 Proof of Correctness of Design of Detector and Corrector Components

We now show the design of the detector and corrector components for the FT-LDP and present the proof of the design of these components.

4.3.1 Design of the Detector Components

Consider the LDP program p, where LDP messages are exchanged between the peers. For ease of exposition, we will assume that the LSR has discovered its peer, initiated the session, and the session is in *Operational (op)* state. Following this, there is an exchange of LDP messages. The intolerant program for LDP, p, is shown below, where *state* denotes the state of the session between the peers, *hello_adj* operation is *true* if there is an hello adjacency between the two LSRs, $no\_of\_pkts\_rxed$ is the number of messages received since the last *KeepAlive* was sent, rx_ka_to indicates the expiry of the KeepAlive timer, $send(i) = false$ and $receive(j) = false$ denote the

failure of the $send()$ and $receive()$ operations.

$p :: true \rightarrow \forall i, j \in processors,$

$(state = op \wedge hello\_adj\ () \wedge rx\_ka\_to \wedge (send\ (i) = true)) \Rightarrow$

$(receive\ (j) = true) \wedge (no\_of\_pkts\_rxed > 0)$

We consider the TCP connection failure, wherein the TCP connection between the peers fails. In the presence of this fault, fail-safe tolerance can be achieved by the following programs, $d1$ and $d2$.

$d1 :: \forall i, j \in processors,$

$(state = op \wedge hello\_adj\ () \wedge \neg(send\ (i) = true) \wedge \neg(tcp\_cx\_failure = true) \Rightarrow$

$\neg(receive\ (j) = true) \rightarrow tcp\_cx\_failure := true$

$d2 :: \forall i, j \in processors,$

$(state = op \wedge hello\_adj\ ()) \wedge rx\_ka\_to) \wedge \neg(tcp\_cx\_failure = true) \Rightarrow$

$\neg(no\_of\_pkts\_rxed > 0) \rightarrow tcp\_cx\_failure := true$

From the definition of detector in [19], a detector is used to check whether the "detection predicate", is true. The detection predicates of the two detectors are $X1$ and $X2$, where $X1$ is $\neg(send(i) = true)$) and $X2$ is $(no\_of\_pkts\_rxed > 0)$. A detection predicate, X, is one for which execution of an action in any state where X is $true$ maintains

the specification. The witness predicate Z in these two cases is $tcp\_cx\_failure = true$. Initially $tcp\_cx\_failure$ is $false$, to indicate that there is no failure. On detection of a failure, namely, $\neg(send(i) = true)$ or $(\neg(no\_of\_pkts\_rxed > 0) \wedge rx\_ka\_to)$, Z is set to *true* to indicate the failure. It satisfies the conditions of *Safety*, since $Z \Rightarrow X1$ and $Z \Rightarrow X2$, *Progress*, since at any state where $X1$ or $X2$ is true, Z is enabled either at that state or somewhere in the future, i.e., the detector eventually detects the fault, and *Stability* is satisfied since Z remains *true* till the recovery procedure resets it. In addition to this, the detection predicates mentioned above hold atomically, i.e., by executing at most one action of the detector.

4.3.2 Design of the Corrector Components

Problem Specification: For the handshaking procedure, the sender sends a sequence numbered message to the receiver, which, upon receiving this message, sends an acknowledgement to the sender, which enables the sender to transmit the next message. A message from the sender has to be sent, one message at a time to the receiver. The sender and receiver communicate using a bidirectional channel (TCP connection) that can hold one message in each direction at a time. The requirement is that each message sent is received at the receiver in the same order as sent.

The message transfer is subject to faults that loose the messages in the channel. Now, we have to design a corrector action which ensures that messages are not lost in the event of a communication failure.

The actions of the program for handshaking includes sending an acknowledgement to the sender, for each message received by the receiver. This can be indicated by the following four actions.

$send\_msg :: rs = true \rightarrow rs := false \wedge mod\_send() \Rightarrow appendtxch(seq)$

By this action, the sender sends a message to the receiver. Since $rs = true$, the sender can transmit a message, which is indicated by the *mod_send* operation, and resetting the rs to $false$, which results in a message with sequence number *seq_no* being appended in the transmit message queue.

$rx\_ack :: \neg(getrxch(msg\_q)) \rightarrow rs, seq := true, seq + 1 \wedge assignrxch()$

This action is executed at the sender when it receives a acknowledgement from the receiver. $\neg(getrxch())$, means there is an acknowledgement sent by the receiver and hence rs is set to $true$, $assignrxch()$ updates the receiver queue, and the sequence number *seq*, at the sender is incremented.

$rx\_msg :: \neg(gettxch(msg\_q)) \rightarrow rr := true \wedge assigntxch() \wedge setNR()$

This action is executed at the receiver when it receives a message from the sender. $\neg(gettxch(msg\_q))$, means there is a message sent by the sender and hence rr is set to $true$, the transmit queue is updated by $assigntxch()$ and the ACK sequence number, at the receiver is updated with the sequence number in the message sent by the sender, using $setNR()$.

$$send\_ack :: rr = true \rightarrow rr := false \wedge mod\_send() \Rightarrow appendrxch(ack\_seq\_no)$$

By this action, the receiver sends an acknowledgement to the sender. Since $rr = true$, the receiver transmits the acknowledgement, which is indicated by the mod_send operation, and resetting the rr to $false$, which results in an acknowledgement with sequence number ack_seq_no being appended in the receive message queue.

Arora [2] proposed a method for designing non-masking fault tolerance based on which we design the corrector component. It puts forth the following design steps to design corrector actions so that the program satisfies the given problem specification and is nonmasking tolerant[5], to a given set of fault actions:

- Designing the invariant S, in which a state predicate S is constructed which is strong enough so that the safety properties of the problem specification are met.
- Designing a fault-span T, in which a state predicate T is constructed which is

[5]In our case study, since we are adding correctors to a program that is composed with detectors, the resulting program will result in a masking tolerant program

weak enough so that the fault actions preserve it.

- Designing program actions that achieve non-masking tolerance by ensuring that T converges to S, meaning that each of these actions preserve T as well as S.

Invariant. When a message is received, $seq = ack\_seq\_no$ is *true*, and remains so until the sender receives an acknowledgement. On reception of an acknowledgement, $ack\_seq\_no = seq - 1$, and this remains *true* till the receiver receives the next message. If the tranmit queue is not empty as indicated by $\neg(gettxch(msg\_q))$, it consists of exactly one message, with sequence number *seq*, and in any state, only one of the four actions is enabled. Thus, the invariant of the program is, S_{ID}, where

---

$$S_{ID} = ((rr = true \vee \neq(getrxch(msg\_q))) \Rightarrow ack\_seq\_no = seq) \wedge$$

$$((rs = true \vee \neq(gettxch(msg\_q))) \Rightarrow ack\_seq\_no = seq - 1) \wedge$$

$$((gettxch(msg\_q)) \vee (gettxch(msg\_q)) = seq) \wedge$$

$$(gettxch(msg\_q) + getrxch(msg\_q) + rs + rr = 1)$$

Fault Actions : The fault action, in this case, failure of the communication channel, could cause the loss of a message sent from sender to receiver or an acknowledgement sent from receiver to the sender. These actions are shown as:

- $\neg(gettxch(msg\_q)) \rightarrow assigntxch()$
- $\neg(getrxch(msg\_q)) \rightarrow assignrxch()$, and
- $tcp\_cx\_failure = true$

The LDP program with handshaking deadlocks when a message or an acknowledgement is lost because of the fault. To add non-masking fault-tolerance, we add a corrector whose correction predicate is S_{ID}. To achieve this correction, the corrector retransmits the last message, when a message or an acknowledgement is lost. As explained in Section 5.5.1 of [15], the receiver processes any re-issued message as if it received it for the first time. Therefore the final action is that of the corrector, which is executed when both the send and receive channels, are empty and rs and rr are both $false$, and the $tcp\_cx\_failure$ is $true$, and it reinitializes the protocol by opening a new TCP connection and retransmitting the message with sequence number, seq. The corrector action is represented as:

$Reissue :: (gettxch\ (msg\_q)) \wedge (getrxch\ (msg\_q)) \wedge (rs, rr = false) \wedge$

$(tcp\_cx\_failure = true) \Rightarrow ReInitialize() \wedge appendtxch(seq)$

Fault – span and Invariant: If a message or acknowledgement is lost, the program reaches a state where transmit and receive channels are empty and rs and rr are $false$. Also, in the presence of faults, if transmit channel is non-empty, it consists of exactly one message whose sequence number is seq. Thus, the fault-span of the non-masking program is

$T_{ND} = (gettxch(msg\_q)) \vee (gettxch(msg\_q) = seq) \wedge (gettxch(msg\_q) + getrxch(msg\_q) + rs + rr \leq 1) \wedge \neg(tcp\_cx\_failure = true)$

and the invariant of the non-masking solution is the same as S_{ID}, i.e.,

S_{ND}=S_{ID}.

Since we are adding the corrector to a fail-safe program (obtained after the addition of Detectors $D1$ and $D2$), the resulting program after the addition of the corrector is masking-tolerant.

Remarks : We have identified the fault-span predicate for the LDP program that is preserved in the presence of TCP connection failure, followed by the design of the invariant which ensures that the safety properties of the LDP specification are met. Finally, we design the program actions that ensure that both fault-span predicate and invariant are preserved, which certifies that the problem specification is fulfilled in any computation that starts from a state where the invariant holds.

4.4 Composing the LDP, Detectors and Correctors using Category Theory

In this section, we explain the composition of the FIP with the fault tolerant components using the concepts of category theory explained in Section 2.4.3. Note that composition is obtained via *union* operation as discussed before.

Figure 4.1 depicts the individual modules of LDP, $D1$ and $D2$. For the composite module LDP_D1_D2 composed out of LDP, $D1$ and $D2$ to exist, the individual modules should commute through the specification morphism as discussed in Sec-

tion 2.4.3. These relationships are shown next to the figure separated by a dashed line.

4.4.1 Composing the LDP with D1 and D2

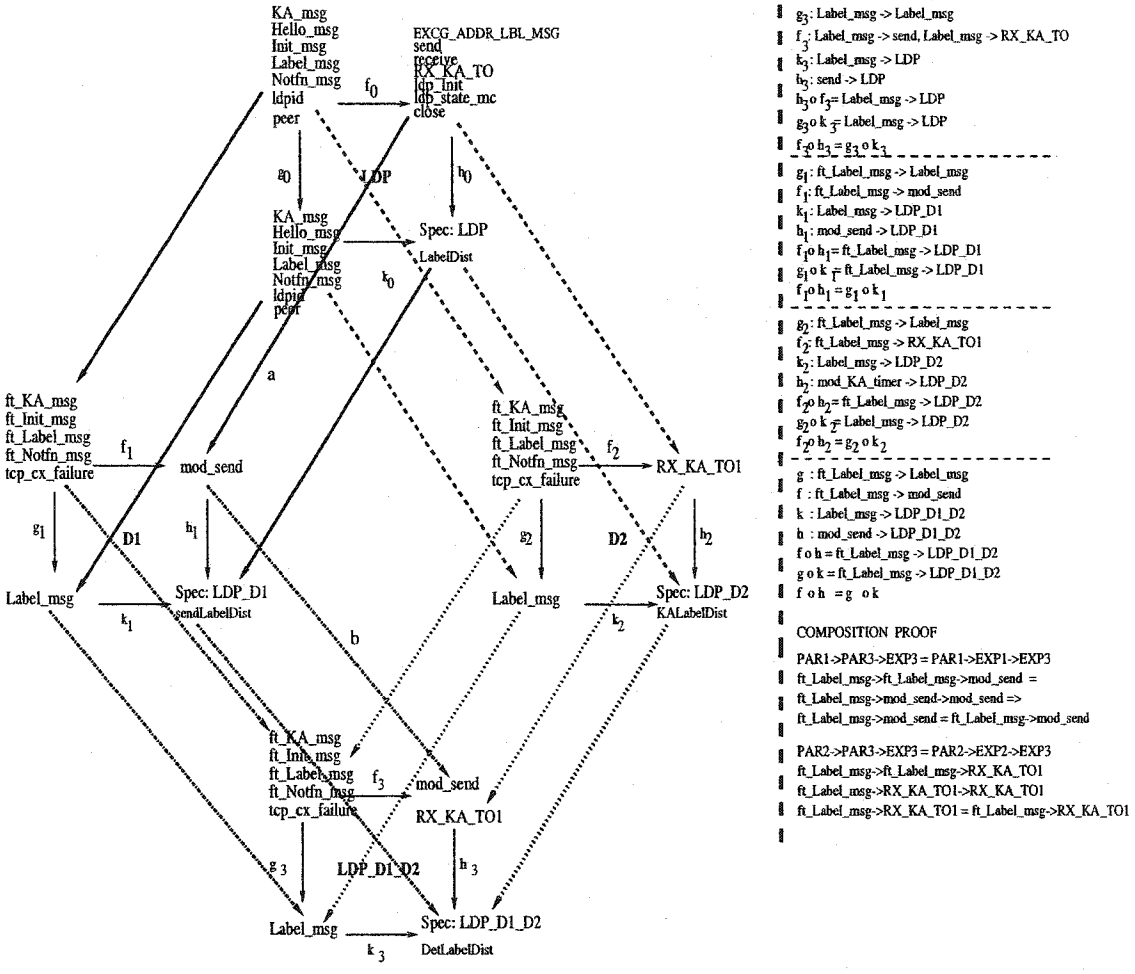

Figure 4.1: Composing the FIP with the Detector Components Using the Union Operation

Each of the modules (components) are depicted in the same way as Figure 2.2 with the four sets of objects as parameters, export, import and body (spec) with corresponding morphisms and mappings. The property of the LDP module is: (a) if A

and B are LSR peers, then a *Label* or *Address* message sent by A is received at B and vice-versa. Since LDP is the basic module, the import interface and parameter part of the specification are equal: $LDP.import = LDP.parameter$ [14]. The LDP module has *ldpid*, *peer*, *self*, *mode*, *Hello_msg*, *Init_msg*, *Label_msg* and *Notification_msg* as its parameters. The export entity consists of the operations that achieve the label distribution property, viz., *SEND_HELLO*, *RX_KA_TO*, *send*, *receive*, *RECEIVE_HELLO*, *ldp_Init*, *ldp_state_mc* and *close*. In the Figure 4.1, f_3 maps the *Label_msg* to the *send* and *RX_KA_TO* operations, i.e., these two operations act on the *Label_msg*, either by sending a Label message using the *send* operation or sending a *KeepAlive* message when the RX_KA_TO timer is triggered. The morphism, h_3 maps the operation *send* to the *LabDist* property in *Spec:LDP* since the operation *send* provides the property of sending Label messages to the peer. The morphism, k_3 maps the variable *Label_msg* to the property *LabDist*, since this variable is being used in providing the final property of communicating label messages between the peer LSRs.

The $D1$ module has the same parameters as that of the LDP module with the addition of the $tcp\_cx\_failure$ and extending the *send* operation as *mod_send*. The property of the $D1$ module is communication of label messages between peer LSRs with the additional property of detecting the TCP connection failure (*sendLabDist*). The $D2$ module also has the same parameters as that of the LDP module with the addition of the $tcp\_cx\_failure$ and extending the RX_KA_TO operation as RX_KA_TO1. The property of the $D2$ module is communication of label messages between peer LSRs

with the additional property of detecting the TCP connection failure (*KALabDist*). The morphisms of these individual modules and the final composed module are shown in the Figure 4.1. The $D1$ and $D2$ modules enhance the existing operations of the LDP module providing the additional functionality of detecting the TCP connection failure. The proof of the composition is also shown in the Figure 4.1.

We have shown how the components, namely, LDP and the detectors $D1$ and $D2$ can be composed using the concepts of category theory, and the correctness is shown in Figure 4.1. The composed module as shown in Figure 4.1 also commutes, and the composition is correct. Hence the combined specification can be further reused for subsequent composition.

4.4.2 Composing the LDP_D1_D2 with C1 and C2

Figure 4.2 depicts the individual modules of LDP_D1_D2, $C1$ and $C2$. For the composite module $LDP\_D1\_D2\_C1\_C2$ composed out of LDP_D1_D2, $C1$ and $C2$ to exist, the individual modules should commute through the specification morphism as discussed in Section 2.4.3. These relationships are shown next to the figure separated by a dashed line.

In this case the composed module obtained after the composition of LDP module with $D1$ and $D2$ forms the base module for composing with the corrector components. Figure 4.2 shows the correctness of construction for each module and the proof of composition. The correctors add the property of restoring the program to its normal

state on detection of the fault.

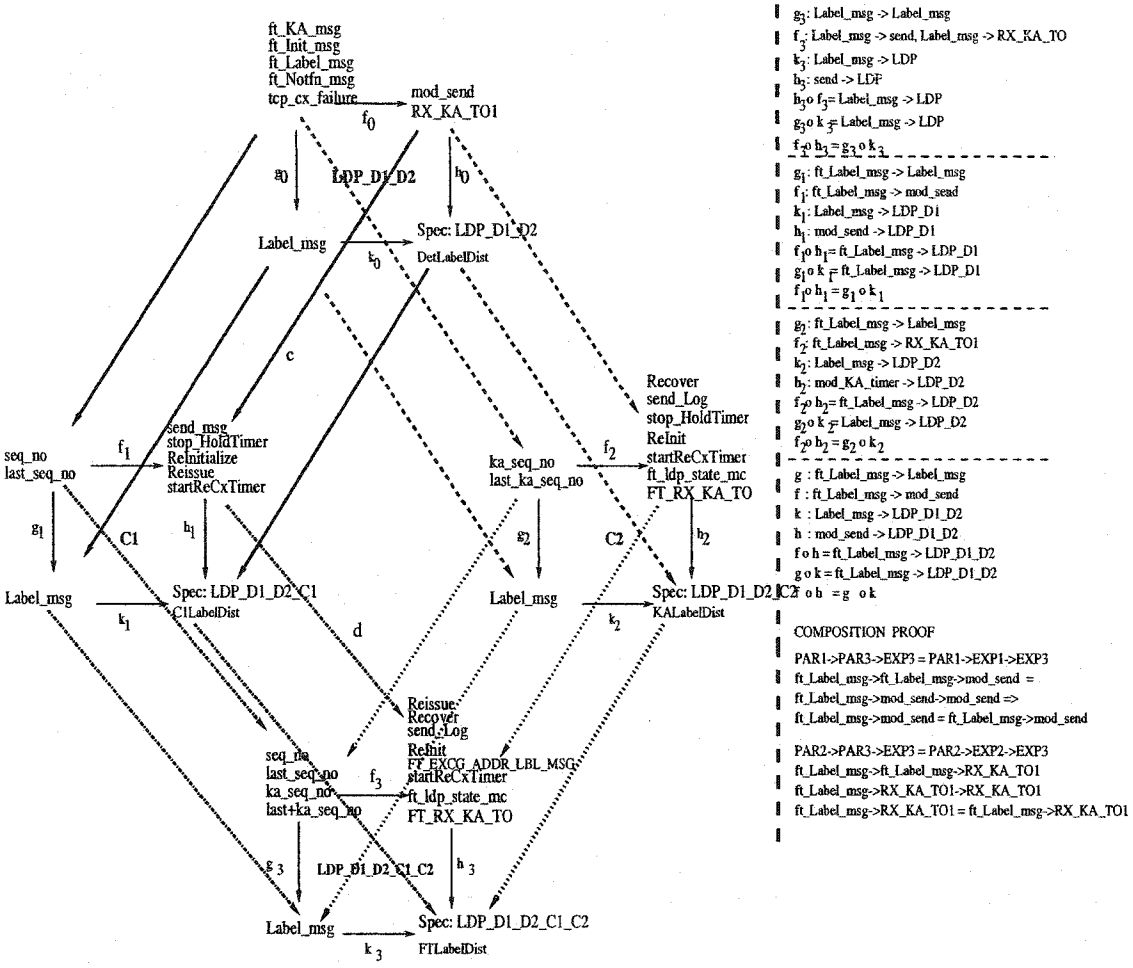

Figure 4.2: Composing the LDP_D1_D2 with the Corrector Components Using the Union Operation

The *LDP_D1_D2* module has *tcp_cx_failure*, *ft_Hello_msg*, *ft_Init_msg*, *ft_Label_msg* and *ft_Notification_msg* as its parameters. The export entity consists of the operations that achieve the label distribution property and detection of the failure, viz., *mod_send* and *RX_KA_TO1*. In the Figure 4.1, g_3 maps the *ft_Label_msg* to the *Label_msg* since the former is a modified version of the *Label_msg* to provide the detection ability, f_3 maps the *ft_Label_msg* to the *mod_send* since this operation acts on

the *ft_Label_msg*, by sending a FT Label message using the *mod_send* operation.

The morphism, h_3 maps the operation *mod_send* to the *DetLabDist* property in *Spec:LDP_D1_D2* since the operation *mod_send* provides the property of sending Label messages to the peer, in addition to detecting the failure. The morphism, k_3 maps the variable *Label_msg* to the property *DetLabelDist*, since this variable is being used in providing the property of communicating label messages between the peer LSRs, in addition to detecting the failure.

The $C1$ module adds the *seq* and *ack_seq_no* parameters to the existing parameters imported from the LDP_D1_D2 module, in addition to the operations shown in the Figure 4.2. The property of the $C1$ module is communication of label messages between peer LSRs in addition to restore the program to its previous state in case of a failure (*C1LabDist*). It achieves this with the *ReInitialize* and *Reissue* operations which make use of the sorts mentioned before. The $C2$ module also has the same parameters as that of the LDP_D1_D2 module with the addition of the *ka_seq_no* and *last_ka_seq_no* parameters and extending the RX_KA_TO1, *ldp_state_mc* in addition to the operations shown in the Figure 4.2. The property of the $C2$ module is recover the program to its normal state when a fault occurs, apart from the communication of label messages between peer LSRs (*C2LabDist*). The morphisms of these individual modules and the final composed module are shown in the Figure 4.2. The $C1$ and $C2$ modules enhance the existing operations of the LDP module providing the additional functionality of correcting the program state on detection of TCP connection failure.

The proof of the composition is also shown in the Figure 4.2. The final composed module therefore has the property of detecting faults and restoring the program to a correct state in addition to the original property of exchanging label messages.

4.5 Compositional Specification and Verification Using Specware

In this section, we provide the specification and verification of the properties for the fault-tolerant Label Distribution Protocol discussed in the previous section. We have used Kestrel's Specware tool [33] for the formal specification and verification of the case study. Specware[6] is a refinement-based approach to software development that supports *rigorous and explicit modularity* in the *specification and development* of software components. There is also a provision for refinement of an abstract specification, primarily done by refining algorithms and data structures. We emphasize that each step in the refinement process is constrained to preserve correctness. We describe the composition starting from the initial fault-intolerant LDP and then composing it with detector and corrector components resulting in the final composed program for label distribution that is 'masking' fault-tolerant.

[6]Details can be found at http://www.specware.org.

4.5.1 Formal Specification of the Label Distribution Protocol

We initially specify the property of the label distribution protocol associated with the following action of "exchanging Address or Label messages(*EXCG_ADDR_LBL_MSG*)."

For exchanging Address or Label messages, the following properties should be met:

- the LDP protocol must be successfully initialized, i.e., the *state* must be *operational*, denoted by the constant *four*, in our formal specification.
- an hello adjacency must exist with the peer with whom address or label messages are to be exchanged,
- an Address or Label message sent must be received by the peer and vice-versa.

This operation is expressed formally as:

```
op EXCG_ADDR_LBL_MSG : msg * my_ldp_id * peer_ldp_id * tx_init * state * address
                       * address * interface * hello_adj_list * msg -> Boolean
axiom EXCG_ADDR_LBL_MSG is fa(I:msg,idc:my_ldp_id,ids:peer_ldp_id,tx:tx_init,
                              s:state,peer:address,se:address,ifc:interface,
                              hl:hello_adj_list,m:msg)
   ldp_Init(I,idc,ids,tx,s,peer,ifc) & hello_adj(hl,peer) &
( SEND_ADDR(m,peer) or SEND_LBL(m,peer))
   => (RECEIVE_ADDR(m,se) or RECEIVE_LBL(m,se)) & (s = 4)
```

A brief explanation of the sorts and operations used is given below:

```
sort interface = Nat      %% the interface on which a packet is sent/received.
sort peer = address       %% the peer address.
sort self = address       %% the address of the node invoking the operation.
sort mode = Boolean       %% the mode of the node "active" means true,
                          %% "passive" means false.
sort my_ldp_id = Nat      %% the LDP identifier of the node invoking the operation.
sort peer_ldp_id = Nat    %% the LDP identifier of the peer.
sort rxr_ldp_id = Nat     %% the LDP identifier received in the Initialization msg.

sort send_hello = Boolean   %% if true => Hello message is to be sent.
sort h_time = Nat           %% value of hold time.
sort msg = Nat              %% General msg for simplicity is indicated by Nat.
sort cr = List msg          %% Receiver channel indicated by a queue of messages.
```

```
sort cs = List msg          %% append(cs,msg,seq_no) -> CS=CSo<NS>;assigncs(msg)
sort hello_ok = Boolean     %% if true => Hello message received is acceptable.
sort allowed_peers = List address    %% list of allowed peers.
sort if_config_list = List interface %% list of interfaces configured for LDP op.
sort hello_adj_list = List address   %% list of neighbours with whom there is
                                     %% a hello adjacency.

sort state = Nat             %% protocol state -- 1-> INITIALIZED,2->OPENSENT,
                             %% 3->OPENREC,4->OPERATIONAL,5->NON_EXISTENT
sort tx_init = Boolean       %% if true => Initialization msg is to be sent.
sort tx_KA = Boolean         %% if true => KeepAlive message is to be sent.
sort no_of_pkts_rxed = Nat   %% indicates number of packets received
sort send_ka_to = Boolean    %% if true => Send KeepAlive timer expired.
sort rx_ka_to = Boolean      %% if true => Receive KeepAlive timer expired.
sort no_of_pkts_sent = Nat   %% indicates number of packets sent.
sort send_hold_to = Boolean  %% if true => Send Hold timer expired.
sort rx_hold_to = Boolean    %% if true => Receive Hold timer expired.
```

The *send_hold_to* and *rx_hold_to* sorts are set to *true* when the send Hold timer and the receive Hold timer expire respectively. The following operations, namely, *SEND_HOLD_TO* and *RECEIVE_HOLD_TO*, reset these sorts after the actions related to the timer operations are done.

```
op reset_send_hold_to : send_hold_to -> Boolean
op reset_rx_hold_to : rx_hold_to -> Boolean
```

The *send_ka_to* and *rx_ka_to* sorts are set to *true* when the send KeepAlive timer and the receive KeepAlive timer expire respectively. The following operations reset these sorts after the actions related to the timer operations are done, namely, *SEND_KA_TO* and *RX_KA_TO*.

```
op reset_send_ka_to : send_ka_to -> Boolean
op reset_rx_ka_to : rx_ka_to -> Boolean
```

The *send_hello* sort is set to *true*, to indicate that a *Hello* message has to be sent. It is reset following the transmission of the *Hello* message. The following operations are used to set and reset the *send_hello* sort respectively.

```
op set_send_hello : send_hello -> Boolean
op reset_send_hello : send_hello -> Boolean
```

The sort *mode* is set to *true*, to indicate that the mode of operation of the protocol is *active*. It is reset to indicate *passive* mode of operation. The following operations are used to set and reset the *mode* sort respectively.

```
op setmode : mode -> Boolean
op resetmode : mode -> Boolean
```

The TCP_CX operation opens a TCP connection with the peer. This is similar to the *connect* system call in socket programming.

```
op TCP_CX : address * address -> Boolean
```

The TCP_ACCEPT operation accepts an incoming TCP connection request from a peer. This is similar to the *accept* system call in socket programming.

```
op TCP_ACCEPT : address * address -> Boolean
```

The $SEND\_NOTFN$ is used to send a Notification message.

```
op SEND_NOTFN : msg * address -> Boolean
```

The following operations are used to send and receive Address messages.

```
op SEND_ADDR : msg * address -> Boolean
op RECEIVE_ADDR : msg * address -> Boolean
```

The *tx_init* sort is set to *true*, to indicate that an *Initialization* message has to be sent. It is reset following the transmission of the *Initialization* message. The following operations are used to set and reset the *tx_Init* sort respectively.

```
op set_txinit : tx_init -> Boolean
op reset_txinit : tx_init -> Boolean
```

The following operation sets the *state* sort to the value specified by *Nat*. This is to keep track of the current state of the protocol as it progresses from *NON_EXISTENT* to *OPERATIONAL* and back to *NON_EXISTENT* on completion of the protocol operation.

```
op setState : state * Nat -> Boolean
```

The *tx_KA* sort is set to *true*, to indicate that a *KeepAlive* message has to be sent. It is reset following the transmission of the *KeepAlive* message. The following operations are used to set and reset the *tx_KA* sort respectively.

```
op set_txKA : tx_KA -> Boolean
op reset_txKA : tx_KA -> Boolean
```

The following operations are used to check if the message is a TCP Connect, Hello, Initialization and KeepAlive message respectively.

```
op isConnect : msg -> Boolean
op isHello : msg -> Boolean
op isInit : msg -> Boolean
op isKA : msg -> Boolean
```

The *close* operation takes care of the closing of LDP session and all other cleanup operations.

```
op close : my_ldp_id -> Boolean
```

The basic *send* and *receive* operations and their axioms are as shown below:

```
op send : msg * address * interface -> Boolean
axiom send is
fa(m:msg,a:address,ifc:interface)
   ~(receive(m,a,ifc)) & send(m,a,ifc)

op receive : msg * address * interface -> Boolean
axiom receive is
fa(m:msg,a:address,ifc:interface)
   ~(send(m,a,ifc)) & receive(m,a,ifc)
```

We now explain the various operations and axioms that implement the steps of the Label Distribution Protocol as explained in Section 4.2.1.

The LDP program provides an interface for an application to initiate the LDP stack using the *ldp_start* operation. This operation takes care of the Discovery mechanism explained in Section 4.2.1. This operation sends the *Hello* message to the peer, and the same is received by the peer, when the *send_hello* sort is set and the peer address is known. This is given as follows:

```
op ldp_start : send_hello * msg * address * interface -> Boolean
axiom ldp_start is fa(b:send_hello,H:msg,p:address,ifc:interface)
   (b = true) & ~(p = 0) => send(H,p,ifc) & receive(H,p,ifc)
```

The *hello_acceptable* operation ensures if a *Hello* message received on an interface from a particular address is valid. If the *Hello* is acceptable, it sets the *hello_ok* sort to true.

```
op hello_acceptable : if_config_list * interface * hello_ok * address *
                      allowed_peers -> Boolean
axiom hello_acceptable is fa(il:if_config_list,ifc:interface,ok:hello_ok,
                             a:address, ap:allowed_peers)
   interface_config(il,ifc) & peer_allowed(ap,a) => (ok = true)
```

The Session Establishment phase mentioned in Section 4.2.1 is taken care of by the *ldp_state_mc* operation. The explanation of the LDP state machine is given in [15]. This operation is formally given below:

```
op ldp_state_mc : msg * state * send_hello * my_ldp_id * tx_init * tx_KA *
                  peer_ldp_id * address * interface -> Boolean
axiom ldp_state_mc is fa (m:msg,s:state,sh:send_hello,myid:my_ldp_id,txi:tx_init,
                          txKA:tx_KA,peer:peer_ldp_id,addr:address,ifc:interface)
   (if ((s=5) & isHello(m)) then
      SEND_HELLO(sh,m,addr,ifc)
   else if ((s=5) & isConnect(m)) then
      SEND_INIT(m,txi,myid,peer,addr,ifc)
```

```
else
   SEND_NOTFN(m,addr) & close(myid) )
or
(if ((s=1) & isHello(m)) then
   SEND_HELLO(sh,m,addr,ifc)
else if ((s=1) & isInit(m)) then
   SEND_INIT(m,txi,myid,peer,addr,ifc)
else
   SEND_NOTFN(m,addr) & close(myid) )
or
(if ((s=2) & isHello(m)) then
   SEND_HELLO(sh,m,addr,ifc)
else if ((s=2) & isInit(m)) then
   SEND_INIT(m,txi,myid,peer,addr,ifc)
else
   SEND_NOTFN(m,addr) & close(myid) )
or
(if ((s=3) & isKA(m)) then
   EXCG_ADDR_LBL_MSG(m,myid,peer,txi,s,addr,self,ifc,hl,m)
else
   SEND_NOTFN(m,addr) & close(myid) )
or
(if ((s=4) & isKA(m)) then
   SEND_HELLO(sh,m,addr,ifc)
else if ((s=4) & ( isInit(m) or isKA(m) )) then
   SEND_KA(m,txKA,addr,ifc)
else
   SEND_NOTFN(m,addr) & close(myid) )
```

The Session Closure phase is ensured by the *close* operation explained earlier. The various operations and axioms needed for these three phases are explained below.

On receiving a *Hello* message, the $RECEIVE\_HELLO$ operation resets the hold timer and if a hello adjacency exists, and the peer address is lesser than this node's address, the mode is set to *active* and a TCP connection is opened with the peer. If the peer address is greater, the mode is set to *passive* and the node waits for the peer to initiate a TCP connection. If an hello adjacency does not exist, then an hello adjacency is created and a Hello message is sent to the peer to acknowledge the hello adjacency, and the mode is set as explained earlier.

```
op RECEIVE_HELLO : hello_adj_list * send_hello * address * address * interface *
                   hello_ok * mode * msg * send_hold_to * rx_hold_to -> Boolean
axiom RECEIVE_HELLO is fa (hl:hello_adj_list,b:send_hello,s:address,p:address,
                           i:interface,ok:hello_ok,m:mode,ms:msg,
                           sho:send_hold_to,rho:rx_hold_to)
   receive(ms,p,i) & reset_rx_hold_to(rho) &
   ( if (hello_adj(hl,p)) then
      (if (s > p) then
            setmode(m) & TCP_CX(s,p)
         else
            resetmode(m) & TCP_ACCEPT(s,p))
      else
      (create_hello_adj(hl,p) & set_send_hello(b) &
         (if (s > p) then
               setmode(m) & TCP_CX(s,p)
            else
               resetmode(m) & TCP_ACCEPT(s,p))
      => SEND_HELLO(b,ms,p,i)))
      => rx_hold_timer(rho)
```

On initialization, the *ldp_Init* operation sets the *tx_init* sort, which inturn ensures that an *Initialization* message is sent to the peer and the *state* is set to *OPENSENT*. This is represented formally as shown below:

```
op ldp_Init : msg * my_ldp_id * peer_ldp_id * tx_init * state * address *
              interface -> Boolean
axiom ldp_Init is fa (I:msg,idc:my_ldp_id,ids:peer_ldp_id,tx:tx_init,s:state,
                      a:address,i:interface)
   ((s = 1) & set_txinit(tx)) => SEND_INIT(I,tx,idc,ids,a,i) & setState(s,2)
```

On receiving an *Initialization* message, if the state is *OPENSENT* and the LDP identifier received in the message is the same as that of the receiving node, then a *KeepAlive* message is sent and the state is set to *OPENREC*. If this is not true, then a *Notification* message is sent and the state is set to *NON_EXISTENT*. The formal representation of this operation is shown below:

```
op RECEIVE_INIT : msg * msg * msg * rxr_ldp_id * state * my_ldp_id * tx_KA *
                  address * interface -> Boolean
axiom RECEIVE_INIT is fa (I:msg,N:msg,K:msg,id:rxr_ldp_id,s:state,idc:my_ldp_id,
                          ka:tx_KA,a:address,i:interface)
```

```
if ( (s = 2) & (id = idc) ) then
   set_txKA(ka) => SEND_KA(K,ka,a,i) & setState(s,3)
else
   SEND_NOTFN(N,a) & setState(s,5)
```

The operation *SEND_KA* is used to send a *KeepAlive* message to the peer. On receiving a *KeepAlive* message, if the state is *OPENREC* it is set to *OPERATIONAL*. The formal representation of the send and receive operations is shown below:

```
op SEND_KA : msg * tx_KA * address * interface -> Boolean
axiom SEND_KA is fa(K:msg,b:tx_KA,a:address,i:interface)
   (b = true) & send(K,a,i) => reset_txKA(b)

op RECEIVE_KA : msg * state -> Boolean
axiom RECEIVE_KA is fa(k:msg,s:state)
   (s = 3) => setState(s,4)
```

We have two timers for the *KeepAlive* message, one for sending a *KeepAlive* message if no message is sent for the KeepAlive period. This is represented by the operation *SEND_KA_TO*. If the number of packets sent is zero and send KeepAlive timer expires, a *KeepAlive* message is sent to the peer. The formal representation of the send timer is shown below:

```
op SEND_KA_TO : msg * state * send_ka_to * no_of_pkts_sent * tx_KA * address *
                interface -> Boolean
axiom SEND_KA_TO is fa (K:msg,s:state,to:send_ka_to,pkt:no_of_pkts_sent,tx:tx_KA,
                        a:address,i:interface)
   (pkt=0) & (to=true) =>
   set_txKA(tx) & SEND_KA(K,tx,a,i) & reset_send_ka_to(to) & (no_of_pkts_rxed=0)
```

The *RX_KA_TO* operation is used to ensure that a message is received in the KeepAlive time period, if not, it is assumed that the connection with the peer is lost or the session no longer exists. If the receive timer times out which is indicated by the *rx_ka_to* and if no packets were received during that interval the state is set to *NON_EXISTENT* and session is closed. This is represented formally as:

```
op RX_KA_TO : msg * state * rx_ka_to * no_of_pkts_rxed -> Boolean
axiom RX_KA_TO is fa(K:msg,s:state,to:rx_ka_to,pkt:no_of_pkts_rxed)
   (to=true) & (pkt = 0) => setState(s,5) & reset_rx_ka_to(to)
```

The translation provided by the *ldp* module which will be used by other modules for their operation is shown below:

```
LDP_to_ALL_TRANSLATION = translate(ldp) by
                         { SEND_ADDR +-> SEND_ADDR, RECEIVE_ADDR+->RECEIVE_ADDR,
                           RX_KA_TO+->RX_KA_TO, send+->send, receive+->receive,
                           EXCG_ADDR_LBL_MSG +-> EXCG_ADDR_LBL_MSG,
                           no_of_pkts_rxed +-> no_of_pkts_rxed,state+->state }
```

We extend the *ldp* specification and add a new variable *tcp_cx_failure* that shall be used by the detector modules which will be added to detect the TCP connection failure. The translation of this new module called *LDP_EXP* is shown below:

```
LDPEXP_to_ALL_TRANSLATION = translate(LDP_EXP) by
                            { tcp_cx_failure +-> tcp_cx_failure }
```

4.5.2 Composing the LDP with the Detector $D1$

The detector $D1$ enhances the *send* operation by setting the *tcp_cx_failure* to *true*, if it sends a message and the peer does not receive it, and the state is *operational* and a hello adjacency exists with the peer. The *mod_send* operation is represented as:

```
op mod_send : tcp_cx_failure * msg * address * interface * state *
              hello_adj_list * address -> Boolean
axiom mod_send is fa (cx:tcp_cx_failure,m:msg,a:address,ifc:interface,s:state,
                      hl:hello_adj_list,peer:address)
   (cx=true) => (s=4) & hello_adj(hl,peer) & send(m,a,ifc) & ~(receive(m,a,ifc))
```

The translation provided by the $D1$ module is given below:

```
LDPD1_to_ALL_TRANSLATION = translate(D1) by { mod_send +-> mod_send }
```

The morphism between the LDP_EXP and the $D1$ specification is formalized as:

```
LDP_TO_D1_MORPHISM = morphism LDP_EXP -> D1 {send +-> send, state +-> state }
```

4.5.3 Composing the LDP with the Detector $D2$

The detector $D2$ enhances the RX_KA_TO operation and sets the $tcp\_cx\_failure$ to true to indicate failure.

```
op RX_KA_TO1 : msg * state * rx_ka_to * no_of_pkts_rxed * tcp_cx_failure *
               hello_adj_list * address -> Boolean
axiom RX_KA_TO1 is fa (m:msg,s:state,to:rx_ka_to,pkt:no_of_pkts_rxed,
                       cx:tcp_cx_failure,hl:hello_adj_list,a:address)
   RX_KA_TO(m,s,to,pkt) & (s=4) & hello_adj(hl,a) & (pkt=0) =>
   reset_rx_ka_to(to) & (cx = true)
```

The translation provided by the $D2$ module is given below:

```
LDPD2_to_ALL_TRANSLATION = translate(D2) by { RX_KA_TO1 +-> RX_KA_TO1 }
```

The morphism between the LDP_EXP and the $D2$ specification is formalized as:

```
LDP_TO_D2_MORPHISM = morphism LDP_EXP -> D2{RX_KA_TO +-> RX_KA_TO,state +-> state}
```

Union to Generate a Composition of LDP Module with Two Dectectors

We then define the diagram with the LDP_EXP, $D1$ and $D2$ specifications as the nodes and the corresponding morphisms as the links between them. This is formally represented as:

```
ldp_D1_D2 = diagram {
   A +-> LDP_EXP,
   B +-> D1,
   m : A -> B +-> morphism LDP_EXP -> D1 { send +-> send },
   C +-> D2,
   i : A -> C +-> morphism LDP_EXP -> D2 { RX_KA_TO +-> RX_KA_TO }
}
```

We finally construct the composite specification of the LDP_EXP, $D1$ and $D2$ modules by taking the co-limit of the diagram, which gives the union of the three specifications. The co-limit operation is specified as :

```
ldpD1D2 = colimit ldp_D1_D2
```

The translation provided by the composed module which will be used by other modules for their operation is shown below:

```
LDPD1D2_to_ALL_TRANSLATION = translate(ldpD1D2) by
                             { mod_send +-> mod_send, RX_KA_TO1 +-> RX_KA_TO1 }
```

We extend the composed specification and add the common sorts that are to be used by both the corrector modules ($C1$ and $C2$), namely, FTP_tlv and $ReCx\_tlv$. These two sorts indicate the S-bit in the "FT Session TLV" if the fault-tolerant operations are to be used and the "FT Reconnect Flag", which denotes whether the LSR has been able to preserve label state respectively. The translation of this new specification called LDP_EXP is shown below:

```
LDPD1D2EXP_to_ALL_TRANSLATION = translate(LDPD1D2_EXP) by
                                { FTP_tlv +-> FTP_tlv, ReCx_tlv +-> ReCx_tlv }
```

4.5.4 Composing Corrector $C1$ with the Composed Module of LDP, $D1$ and $D2$

The corrector $C1$ adds the following sorts *seq*, for identifying the message to be acknowledged, *ack_seq_no*, for the corresponding acknowledgement, and *msq_q* to indicate the queue of channel messages, both at the sender and receiver.

The main operations of the corrector $C1$ are:

```
op send_msg : FTP_tlv * ReCx_tlv * rs * seq * tcp_cx_failure * msg * address *
              interface * state * hello_adj_list-> Boolean
axiom send_msg is fa (ftlv:FTP_tlv,rtlv:ReCx_tlv,r:rs,ns:seq,tcp:tcp_cx_failure,
                      m:msg,a:address,i:interface,s:state,hl:hello_adj_list)
   (ftlv = true) & (rtlv = true) & (r = true) =>
   resetRxAck(r) & mod_send(ftlv,tcp,m,a,i,s,hl,a) => appendtxch(ns)
```

This operation is the action that takes place at the sender when it sends a message to the receiver. If the sort *rs* is *true*, it implies that sender has received an acknowledgement for the last message it sent, and hence it is reset and the next message is sent which is indicated by appending the message to the transmit message queue.

```
op rx_ack : FTP_tlv * ReCx_tlv * rs * seq * msg_q * Nat -> Boolean
axiom rx_ack is fa(ftlv:FTP_tlv,rtlv:ReCx_tlv,r:rs,ns:seq,rx_q:msg_q,n:Nat)
   (ftlv = true) & (rtlv = true) & ~(getrxch(rx_q)) =>
   setRxAck(r) & assignrxch(nth(rx_q,n)) & setNS(ns+1)
```

This operation is the action that takes place at the sender when it receives an acknowledgement from the receiver for a message it sent. The sort *rs* is set to *true* to indicate this, and the receive message queue is updated and the sender's sequence number is incremented, which is tagged with the next message to be sent.

```
op rx_msg : FTP_tlv * ReCx_tlv * rr * ack_seq_no * msg_q * Nat -> Boolean
axiom rx_msg is fa(ftlv:FTP_tlv,rtlv:ReCx_tlv,r:rr,nr:ack_seq_no,tx_q:msg_q,n:Nat)
   (ftlv = true) & (rtlv = true) & ~(gettxch(tx_q)) =>
   setTxMsg(r) & assigntxch(nth(tx_q,n)) & setNR(getAckSeq(hd(tx_q)))
```

This operation is the action that takes place at the receiver when it receives a message from the sender. The sort *rr* is *true*, which indicates that the receiver has received a message, and the transmit message queue is updated.

```
op send_ack : FTP_tlv * ReCx_tlv * rr * ack_seq_no * tcp_cx_failure * msg *
              address * interface * state * hello_adj_list -> Boolean
axiom send_ack is fa (ftlv:FTP_tlv,rtlv:ReCx_tlv,r:rr,nr:ack_seq_no,
                      tcp:tcp_cx_failure,m:msg,a:address,i:interface,
                      s:state,hl:hello_adj_list)
   (ftlv = true) & (rtlv = true) & (r = true) =>
   resetTxMsg(r) & mod_send(ftlv,tcp,m,a,i,s,hl,a) => appendrxch(nr)
```

This operation is the action that takes place at the receiver when it sends a acknowledgement to the sender, which is indicated by the sort *rr* being *true*. The consequent actions reset this sort and the next message is appended in the receive channel queue.

```
op Reissue : FTP_tlv * ReCx_tlv * rr * rs * seq * msg_q * msg_q * my_ldp_id
             -> Boolean
axiom Reissue is fa (ftlv:FTP_tlv,rtlv:ReCx_tlv,r1:rr,r2:rs,ns:seq,tx_q:msg_q,
                     rx_q:msg_q,my:my_ldp_id)
   (ftlv = true) & (rtlv = true) & gettxch(tx_q) & getrxch(rx_q) & (r1 = false)
   & (r2 = false) => ReInitialize(my) & appendtxch(ns)
```

This operation ensures that if the transmit and receive channels are empty and the sorts *rs* and *rs* are *false*, then the sender retransmits the last message.

The translation provided by the $C1$ module is given below:

```
LDPD1D2C1_to_ALL_TRANSLATION = translate(C1) by
                               { send_msg +-> send_msg,rx_ack +-> rx_ack,
                                 rx_msg +-> rx_msg,send_ack +-> send_ack,
                                 Reissue +-> Reissue }
```

The morphism between the $LDPD1D2\_EXP$ and the $C1$ specification is formalized as :

```
LDPD1D2_TO_C1_MORPHISM = morphism LDPD1D2_EXP -> C1 {send +-> send,state +-> state}
```

4.5.5 Composing Corrector $C2$ with the Composed Module of LDP, $D1$ and $D2$

The corrector $C2$ adds the following sorts *kaseq_no*, to indicate the current *KeepAlive* sequence number, *last_kaseq_no*, to indicate the last sent KeepAlive sequence number, *record* to indicate the information that is being flushed, and *chkpt_tlv*, to indicate if checkpointing is to be used.

The operations of $C2$ are *writeLog* for logging sent messages, $flush$, for flushing the log, *Recover* for recovering from the last checkpointed state.

```
op writeLog : msg -> Boolean
op flush : record -> Boolean
op Recover : FTP_tlv * ReCx_tlv -> Boolean
```

The other operations and axioms of the corrector $C2$ include:

```
op sendLog : FTP_tlv * ReCx_tlv * chkpt_tlv * tcp_cx_failure * msg * address *
             interface * state * hello_adj_list -> Boolean
axiom sendLog is fa (ft:FTP_tlv,re:ReCx_tlv,ck:chkpt_tlv,tcp:tcp_cx_failure,
                     m:msg,a:address,i:interface,s:state,hl:hello_adj_list)
   if (ft & re & ck) then
      writeLog(m) & mod_send(tcp,m,a,i,s,hl,a)
   else
      mod_send(tcp,m,a,i,s,hl,a)
```

This operation writes a message to the log before sending it, if the checkpointing procedures are enabled.

```
op chkpoint : FTP_tlv * ReCx_tlv * chkpt_tlv * record -> Boolean
axiom chkpoint is fa(ft:FTP_tlv,re:ReCx_tlv,ck:chkpt_tlv,rec:record)
   (ft = true) & (re = true) & (ck = true) => flush(rec)
```

This operation flushes the messages that have been logged.

```
op ReInit : FTP_tlv * ReCx_tlv * chkpt_tlv * last_kaseq_no * tcp_cx_failure *
            kaseq_no * msg * address * interface * state * hello_adj_list *
            recx_to * rx_hold_to -> Boolean
axiom ReInit is fa (ft:FTP_tlv,re:ReCx_tlv,ck:chkpt_tlv,lka:last_kaseq_no,
                    tcp:tcp_cx_failure,ka:kaseq_no,m:msg,a:address,i:interface,
                    s:state,hl:hello_adj_list,recx:recx_to,rxho:rx_hold_to)
   if((lka < ka) & (tcp = true)) then
      startReCxTimer(recx) & reset_rx_hold_to(rxho) & Recover(ft,re)
   else
      sendLog(ft,re,ck,tcp,m,a,i,s,hl)
```

This operation is invoked when a new TCP connection is opened and the failed session needs to be restored. This operation takes care of reinitializing the protocol to a state where there are no lost or unacknowledged messages. This is ensured by the *Recover* operation.

```
op FT_RX_KA_TO : msg * state * rx_ka_to * no_of_pkts_rxed * tcp_cx_failure *
                 hello_adj_list * address * FTP_tlv * ReCx_tlv * chkpt_tlv *
                 record -> Boolean
axiom FT_RX_KA_TO is fa (m:msg,s:state,to:rx_ka_to,pkt:no_of_pkts_rxed,
                         cx:tcp_cx_failure,hl:hello_adj_list,a:address,
                         ft:FTP_tlv,re:ReCx_tlv,ck:chkpt_tlv,rec:record)
   RX_KA_TO1(m,s,to,pkt,cx,hl,a) & chkpoint(ft,re,ck,rec)
```

This operation enhances the RX_KA_TO1 operation by checkpointing when the KeepAlive timer expires.

```
op ft_ldp_state_mc : msg * state * send_hello * my_ldp_id * tx_init * tx_KA *
                     peer_ldp_id * address * address * hello_adj_list *
                     interface * FTP_tlv * ReCx_tlv * chkpt_tlv * record
                     -> Boolean
axiom ft_ldp_state_mc is fa (m:msg,s:state,sh:send_hello,myid:my_ldp_id,
                             txi:tx_init,txKA:tx_KA,peer:peer_ldp_id,addr:address,
                             self:address,hl:hello_adj_list,ifc:interface,
                             ftlv:FTP_tlv,recx:ReCx_tlv,ckpt:chkpt_tlv,rec:record)
   if((s=4) & (ftlv=true) & (ckpt=true) & (recx=true) & isKA(m)) then
      chkpoint(ftlv,recx,ckpt,rec)
   else
      ldp_state_mc(m,s,sh,myid,txi,txKA,peer,addr,self,hl,ifc)
```

This operation modifies the *ldp_state_mc* such that when the *state* is *Operational* and a *KeepAlive* message is received, the state information is checkpointed.

The translation provided by the $C2$ module is given below:

```
LDPC2_to_ALL_TRANSLATION = translate(C2) by
                           { sendLog +-> sendLog, chkpoint +-> chkpoint,
                             ReInit +-> ReInit }
```

The morphism between the $LDPD1D2\_EXP$ and the $C2$ specification is formalized as :

```
LDPD1D2_TO_C2_MORPHISM = morphism LDPD1D2_EXP -> C2
                         { send +-> send, state +-> state }
```

Union to Generate the Final Composed Module of LDP with both Detectors and Correctors

We then define the diagram with the $LDPD1D2\_EXP$, $C1$ and $C2$ specifications as the nodes and the corresponding morphisms as the links between them. This is formally represented as:

```
ldpD1D2_C1_C2 = diagram {
   A +-> LDPD1D2_EXP,
   B +-> C1,
   m : A -> B +-> morphism LDPD1D2_EXP -> C1 { send +-> send },
   C +-> C2,
   i : A -> C +-> morphism LDPD1D2_EXP -> C2 { RX_KA_TO +-> RX_KA_TO }
}
```

We finally construct the composite specification of the $LDPD1D2\_EXP$, $C1$ and $C2$ modules by taking the co-limit of the diagram, which gives the union of the three specifications. The co-limit operation is specified as :

```
ldpD1D2C1C2 = colimit ldpD1D2_C1_C2
```

The translation provided by the composed module which will be used by other modules for their operation is shown below:

```
LDPD1D2C1C2_to_ALL_TRANSLATION = translate(ldpD1D2C1C2) by
                                 { EXCG_ADDR_LBL_MSG +-> EXCG_ADDR_LBL_MSG }
```

The final theorem that is to be proved from the composed specification is:

```
theorem FT_EXCG_ADDR_LBL_MSG is fa (I:msg,idc:my_ldp_id,ids:peer_ldp_id,tx:tx_init,
                                    s:state,peer:address,se:address,ifc:interface,
                                    hl:hello_adj_list,A:msg,ftv:FTP_tlv,rec:ReCx_tlv,
                                    ckpt:chkpt_tlv,rx:rs,seq:seq,txq:msg_q,n:Nat,
                                    ack:ack_seq_no,rxq:msg_q,tcp:tcp_cx_failure,
                                    ck:chkpt_tlv,recd:record,txm:rr,lka:last_kaseq_no,
                                    kaseq:kaseq_no,recx:recx_to,rxho:rx_hold_to)
   (~(EXCG_ADDR_LBL_MSG(I,idc,ids,tx,s,peer,se,ifc,hl,A)) & (tcp = true)) =>
   ( send_msg(ftv,rec,rx,seq,tcp,A,peer,ifc,s,hl) => rx_ack(ftv,rec,rx,seq,rxq,n)
   => rx_msg(ftv,rec,txm,ack,txq,n) => send_ack(ftv,rec,txm,ack,tcp,A,peer,ifc,s,hl)
   => Reissue(ftv,rec,txm,rx,seq,txq,rxq,idc) ) or (ckpt = true) =>
   (chkpoint(ftv,rec,ck,recd) &
   ReInit(ftv,rec,ck,lka,tcp,kaseq,I,peer,ifc,s,hl,recx,rxho))
```

The theorem, *FT_EXCG_ADDR_LBL_MSG* is the final FT exchange of *Address* or *Label* messages, which states that if the *EXCG_ADDR_LBL_MSG* fails because of a TCP connection failure, it leads to the triggering of either corrector $C1$ or $C2$ actions which ensures that a new TCP connection is opened and the program is restored to a consistent state from which *Address* or *Label* messages are again successfully exchanged. We finally verify the above mentioned property by processing the above specification along with the theorem in Specware with a built-in interface to Snark theorem prover. Reusability is shown by the fact that the theorem *FT_EXCG_ADDR_LBL_MSG* uses axioms like *send*, *receive*, *ldp_Init*, *EXCG_ADDR_LBL_MSG*, *mod_send* and *ReInit* from other modules for its proof as shown in the statement below:

```
p=prove FT_EXCG_ADDR_LBL_MSG in ft_ldp using send receive ldp_Init
        EXCG_ADDR_LBL_MSG mod_send ReInit
```

Note that we have axiomatize various properties of underlying protocols, however, it is to be emphasized that these properties must be proven as a theorem in their respective modules for an extensive formal analysis of the protocol under study.

4.5.6 Role of Morphisms for Traceability

In establishing a particular property, it is important to have an ability to trace all the properties that are influencing it. In our framework, we can achieve traceability through morphisms linking various modules and facilitating functionalities across the composition. The role of different modules in establishing the correctness of "exchanging Address and Label messages" under different failure scenarios is explained below.

The property of "exchanging Address and Label messages" is achieved by the $EXCG\_ADDR\_LBL\_MSG$ axiom, which in turn uses the *send*, *receive* and *ldp_Init* operations for its proper working. The property of the $D1$ module is to achieve the detection of the TCP connection failure which is achieved by the *mod_send* operation. The *mod_send* operation which extends the *send* operation maps the original label distribution property to the property of detecting TCP failures in addition to label distribution. This mapping between the *send* and *mod_send* operations is achieved through the morphism "a" as shown in Figure 4.1. The composed module LDP_D1_D2 now combines the property of detecting the TCP failure either through the failure of the *send* operation or the KeepAlive timeout. The mapping

of *mod_send* operation from $D1$ to *mod_send* operation in LDP_D1_D2 is shown through the morphism "*b*" in Figure 4.1. Simlarly, the $C1$ module which provides the correction property uses the *ReInit* operation for restoring the program to its consistent state. This operation in turn uses the *mod_send* operation that the detector module added in the previous composition. This mapping from the LDP_D1_D2 to the $C1$ module is shown by the morphism "*c*" as shown in Figure 4.2. The final operation that achieves the fault-tolerant property of exchanging label and address messages uses *ReInit* for its operation. This mapping is shown by the morphism "*d*" in Figure 4.2. From this discussion, it is clear that morphisms a, b, c and d linking the various modules help in precisely identifying properties that must be observed across different modules, for e.g., when $D1$ gets composed with LDP, detection of the TCP connection failure is achieved, and it gets carried over to the final composed module, FT_LDP, which needs this property for satisfying the required operation of detecting TCP failure. Through this discussion, we have illustrated that with properties being inherited through morphisms over successive module composition, one can trace back all influential properties needed in establishing the correctness of the resulting fault-tolerant program.

In this Chapter, we have shown the decomposition of the FT-LDP into a FIP and fault-tolerant components and shown that these components can be built as detectors and correctors. In the next Chapter, we make conclusions and list future research directions.

Chapter 5

Conclusion

This thesis aims at the component-based construction of fault-tolerant software using fault-tolerant components. The construction of these components is based on the premise that any fault can be represented as a state perturbation. Kulkarni [19] has illustrated this by case studies which deal with various classes of transient faults, permanent faults, detectable and undetectable faults. We base our construction of fault tolerant components on this approach because of the wide range of faults that can be covered by this approach. Furthermore, this approach also shows that the fault-tolerant components themselves are not faulty and do not interfere in the working of the fault-intolerant program. In this Chapter, we first discuss some of the insights obtained from the design of the corrector and detector components for the multimedia protocol in terms of experience gained through their design and their formal proof followed by the contributions of this thesis. Finally we put forth new directions for

further research.

5.1 Contributions

Our main contribution in this thesis has been to introduce an integrated framework that incorporates the component-based design for fault-tolerance and category-theoretic operations for composition through the correctness-by-construction approach. Our specific contributions in this thesis include:

- we have illustrated our approach for composition using the case study in [5] (see Chapter 3). We have shown the feasibility of our approach in this case study by specifying the algorithm and the fault-tolerant components followed by a proof of the composition of these components with the fault-intolerant program.
- we have shown this composition approach for the LDP (see Chapter 4). We first established the correctness of the detectors and correctors designed using this approach and consequently composed these components with the FIP using the category theory approach to obtain the final FTP.
- we have also shown the formal specification for the protocol, which has been written in an algebraic language, namely, *MetaSlang*.

We have made full use of the Specware [33] development system, which provides for a rigor in establishing the overall correctness of the composition, and moreover, facilitates further transformation into executable code. We emphasize the fact that

over the process of designing and formalizing fault-tolerant version of LDP [15] using the component-based approach, subtle intricacies in the design of fault-tolerant components have been highlighted and incorporated in our formal treatment of the protocol.

5.2 Experience

Our aim in this thesis has been to apply our proposed category-theoretical approach for the composition of a fault-tolerant program by integrating a FIP with fault-tolerant components, namely, detectors and correctors. These components are designed to detect or correct a particular fault for a particular problem or algorithm followed by composing them with the fault-intolerant program. We have shown that the resulting composed solution is fault-tolerant to the faults that can be tolerated by these fault-tolerant components. With our first case study, we show the feasibility of our approach following which we design the detector and corrector components for a multimedia protocol, and show their correctness, based on the work of Arora and Kulkarni. We then show the composition of these components using category theory constructs and show that the composed program does indeed preserve its property even in the case of faults.

In order to achieve fault-tolerance we have shown the construction of these fault-tolerant components in Chapter 4. In the development of reusable components, formal methods can help to promote software reuse. Components that have been formally

specified and sufficiently well documented can be identified, reused, and combined in a new system. Also, it is important to focus on the reuse of formally developed specifications as well as formally developed code; as such reuse can improve the generality versus specialization trade-off.

Our future research directions include finding a generic framework of fault-tolerant components for a class of protocols like transaction processing protocols, network and security protocols and so on. We would also like to explore a way of formally arriving at the invariant for a given problem, which is a "creative process" as of now. We also plan on utilizing the automated code-generation feature of the Specware tool by converting the specifications we have written for our case studies into executable code and verifying that the program is indeed fault-tolerant. We also plan to compare the code generated with an implementation of these programs that have been written without being formally specified. We would like to compare both the implementations in terms of code size, memory usage, performance and other factors, which would enable us to get a better understanding of the code generated using automated procedures.

Appendix

Specification of the composite fault-tolerant LDP *ftldp* along with the processing steps

Specware and proof results from Snark.

```
CL-USER(1): :cd Z:\Win2K_Profile\ldp
Z:\Win2K_Profile\ldp\
CL-USER(2): :sw ftldp.sw
Processing spec at Z:/Win2K_Profile/ldp/ftldp#Address
Processing spec at C:/Program Files/Specware4.0/Library/Base
Processing spec at C:/Program Files/Specware4.0/Library/Base/Boolean
Processing spec at C:/Program Files/Specware4.0/Library/Base/PrimitiveSorts
Processing spec at C:/Program Files/Specware4.0/Library/Base/Compare
Processing spec at C:/Program Files/Specware4.0/Library/Base/Functions
Processing spec at C:/Program Files/Specware4.0/Library/Base/Integer
Processing spec at C:/Program Files/Specware4.0/Library/Base/Nat
Processing spec at C:/Program Files/Specware4.0/Library/Base/Char
Processing spec at C:/Program Files/Specware4.0/Library/Base/String
Processing spec at C:/Program Files/Specware4.0/Library/Base/List
Processing spec at C:/Program Files/Specware4.0/Library/Base/Option
Processing spec at C:/Program Files/Specware4.0/Library/Base/System
Processing spec at C:/Program Files/Specware4.0/Library/Base/Show
Processing spec at Z:/Win2K_Profile/ldp/ftldp#ldp
Processing translation at Z:/Win2K_Profile/ldp/ftldp#LDP_to_ALL_TRANSLATION
Processing spec at Z:/Win2K_Profile/ldp/ftldp#LDP_EXP
Processing translation at Z:/Win2K_Profile/ldp/ftldp
#LDPEXP_to_ALL_TRANSLATION
Processing spec at Z:/Win2K_Profile/ldp/ftldp#D1
Processing translation at Z:/Win2K_Profile/ldp/ftldp
#LDPD1_to_ALL_TRANSLATION
Processing spec morphism at Z:/Win2K_Profile/ldp/ftldp#LDP_TO_D1_MORPHISM
```

```
Processing spec at Z:/Win2K_Profile/ldp/ftldp#D2
Processing translation at Z:/Win2K_Profile/ldp/ftldp
#LDPD2_to_ALL_TRANSLATION
Processing spec morphism at Z:/Win2K_Profile/ldp/ftldp#LDP_TO_D2_MORPHISM
Processing spec diagram at Z:/Win2K_Profile/ldp/ftldp#ldp_D1_D2
Processing spec morphism at Z:/Win2K_Profile/ldp/ftldp#ldp_D1_D2
Processing spec morphism at Z:/Win2K_Profile/ldp/ftldp#ldp_D1_D2
Processing colimit at Z:/Win2K_Profile/ldp/ftldp#ldpD1D2
Processing spec at C:/Program Files/Specware4.0/Library/Base
Processing spec at C:/Program Files/Specware4.0/Library/Base/Boolean
Processing spec at C:/Program Files/Specware4.0/Library/Base/PrimitiveSorts
Processing spec at C:/Program Files/Specware4.0/Library/Base/Compare
Processing spec at C:/Program Files/Specware4.0/Library/Base/Functions
Processing spec at C:/Program Files/Specware4.0/Library/Base/Integer
Processing spec at C:/Program Files/Specware4.0/Library/Base/Nat
Processing spec at C:/Program Files/Specware4.0/Library/Base/Char
Processing spec at C:/Program Files/Specware4.0/Library/Base/String
Processing spec at C:/Program Files/Specware4.0/Library/Base/List
Processing spec at C:/Program Files/Specware4.0/Library/Base/Option
Processing spec at C:/Program Files/Specware4.0/Library/Base/System
Processing spec at C:/Program Files/Specware4.0/Library/Base/Show
Processing translation at Z:/Win2K_Profile/ldp/ftldp
#LDPD1D2_to_ALL_TRANSLATION
Processing spec at Z:/Win2K_Profile/ldp/ftldp#LDPD1D2_EXP
Processing translation at Z:/Win2K_Profile/ldp/ftldp
#LDPD1D2EXP_to_ALL_TRANSLATION
Processing spec at Z:/Win2K_Profile/ldp/ftldp#C1
Processing translation at Z:/Win2K_Profile/ldp/ftldp
#LDPD1D2C1_to_ALL_TRANSLATION
Processing spec morphism at Z:/Win2K_Profile/ldp/ftldp#LDPD1D2_TO_C1_MORPHISM
Processing spec at Z:/Win2K_Profile/ldp/ftldp#C2
Processing translation at Z:/Win2K_Profile/ldp/ftldp#LDPC2_to_ALL_TRANSLATION
Processing spec morphism at Z:/Win2K_Profile/ldp/ftldp#LDPD1D2_TO_C2_MORPHISM
Processing spec diagram at Z:/Win2K_Profile/ldp/ftldp#ldpD1D2_C1_C2
Processing spec morphism at Z:/Win2K_Profile/ldp/ftldp#ldpD1D2_C1_C2
Processing spec morphism at Z:/Win2K_Profile/ldp/ftldp#ldpD1D2_C1_C2
Processing colimit at Z:/Win2K_Profile/ldp/ftldp#ldpD1D2C1C2
Processing translation at Z:/Win2K_Profile/ldp/ftldp
#LDPD1D2C1C2_to_ALL_TRANSLATION
Processing spec at Z:/Win2K_Profile/ldp/ftldp#ft_ldp

spec
```

```
sort FTP_tlv = Boolean
sort HELLO = Nat
sort ReCx_tlv = Boolean
sort ack_seq_no = Nat
sort address = Nat
sort allowed_peers = List(address)
sort chkpt_tlv = Boolean
sort cr = List(msg)
sort cs = List(msg)
sort h_time = Nat
sort hello_adj_list = List(address)
sort hello_ok = Boolean
sort if_config_list = List(interface)
sort interface = Nat
sort kaseq_no = Nat
sort last_kaseq_no = Nat
sort mode = Boolean
sort msg = Nat
sort msg_q = List(msg)
sort my_ldp_id = Nat
sort no_of_pkts_rxed = Nat
sort no_of_pkts_sent = Nat
sort peer = address
sort peer_ldp_id = Nat
sort record = Nat
sort recx_to = Boolean
sort rr = Boolean
sort rs = Boolean
sort rx_hold_to = Boolean
sort rx_ka_to = Boolean
sort rxr_ldp_id = Nat
sort self = address
sort send_hello = Boolean
sort send_hold_to = Boolean
sort send_ka_to = Boolean
sort seq = Nat
sort state = Nat
sort tcp_cx_failure = Boolean
sort tx_KA = Boolean
sort tx_init = Boolean
op EXCG_ADDR_LBL_MSG :
   msg *
```

```
   my_ldp_id *
   peer_ldp_id *
   tx_init *
   state *
   address *
   address *
   interface *
   hello_adj_list *
   msg -> Boolean
op FT_RX_KA_TO :
   msg *
   state *
   rx_ka_to *
   no_of_pkts_rxed *
   tcp_cx_failure *
   hello_adj_list *
   address *
   FTP_tlv *
   ReCx_tlv *
   chkpt_tlv *
   record -> Boolean
op RECEIVE_ADDR : msg * address -> Boolean
op RECEIVE_HELLO :
   hello_adj_list *
   send_hello *
   address *
   address *
   interface *
   hello_ok *
   mode *
   msg *
   send_hold_to *
   rx_hold_to -> Boolean
op RECEIVE_INIT :
   msg *
   msg *
   msg *
   rxr_ldp_id *
   state *
   my_ldp_id *
   tx_KA *
   address *
```

```
   interface -> Boolean
op RECEIVE_KA : msg * state -> Boolean
op RECEIVE_LBL : msg * address -> Boolean
op RX_HOLD_TO : msg * rx_hold_to -> Boolean
op RX_KA_TO : msg * state * rx_ka_to * no_of_pkts_rxed -> Boolean
op RX_KA_TO1 :
   FTP_tlv *
   msg *
   state *
   rx_ka_to *
   no_of_pkts_rxed *
   tcp_cx_failure *
   hello_adj_list *
   address -> Boolean
op ReInit :
   FTP_tlv *
   ReCx_tlv *
   chkpt_tlv *
   last_kaseq_no *
   tcp_cx_failure *
   kaseq_no *
   msg *
   address *
   interface *
   state *
   hello_adj_list *
   recx_to *
   rx_hold_to -> Boolean
op ReInitialize : my_ldp_id -> Boolean
op Recover : FTP_tlv * ReCx_tlv -> Boolean
op Reissue :
   FTP_tlv * ReCx_tlv * rr * rs * seq * msg_q * msg_q * my_ldp_id -> Boolean
op SEND_ADDR : msg * address -> Boolean
op SEND_HELLO : send_hello * msg * address * interface -> Boolean
op SEND_HOLD_TO :
   msg * send_hold_to * send_hello * address * interface -> Boolean
op SEND_INIT :
   msg * tx_init * my_ldp_id * peer_ldp_id * address * interface -> Boolean
op SEND_KA : msg * tx_KA * address * interface -> Boolean
op SEND_KA_TO :
   msg * state * send_ka_to * no_of_pkts_sent * tx_KA * address * interface ->
      Boolean
```

```
op SEND_LBL : msg * address -> Boolean
op SEND_NOTFN : msg * address -> Boolean
op TCP_ACCEPT : address * address -> Boolean
op TCP_CX : address * address -> Boolean
op appendrxch : seq -> Boolean
op appendtxch : seq -> Boolean
op assignrxch : msg -> Boolean
op assigntxch : msg -> Boolean
op chkpoint : FTP_tlv * ReCx_tlv * chkpt_tlv * record -> Boolean
op close : my_ldp_id -> Boolean
op create_hello_adj : hello_adj_list * address -> Boolean
op flush : record -> Boolean
op ft_ldp_state_mc :
   msg *
   state *
   send_hello *
   my_ldp_id *
   tx_init *
   tx_KA *
   peer_ldp_id *
   address *
   address *
   hello_adj_list *
   interface *
   FTP_tlv *
   ReCx_tlv *
   chkpt_tlv *
   record -> Boolean
op getAckSeq : msg -> ack_seq_no
op getrxch : msg_q -> Boolean
op gettxch : msg_q -> Boolean
op hello_acceptable :
   if_config_list * interface * hello_ok * address * allowed_peers -> Boolean
op hello_adj : hello_adj_list * address -> Boolean
op inc_no_of_pkts_rxed : no_of_pkts_rxed -> Boolean
op interface_config : if_config_list * interface -> Boolean
op isConnect : msg -> Boolean
op isHello : msg -> Boolean
op isInit : msg -> Boolean
op isKA : msg -> Boolean
op ldp_Init :
   msg * my_ldp_id * peer_ldp_id * tx_init * state * address * interface ->
```

```
      Boolean
op ldp_start : send_hello * msg * address * interface -> Boolean
op ldp_state_mc :
   msg *
   state *
   send_hello *
   my_ldp_id *
   tx_init *
   tx_KA *
   peer_ldp_id *
   address *
   address *
   hello_adj_list *
   interface -> Boolean
op mod_send :
   FTP_tlv *
   tcp_cx_failure *
   msg *
   address *
   interface *
   state *
   hello_adj_list *
   address -> Boolean
op peer_allowed : allowed_peers * address -> Boolean
op receive : msg * address * interface -> Boolean
op resetRxAck : rs -> Boolean
op resetTxMsg : rr -> Boolean

op reset_rx_hold_to : rx_hold_to -> Boolean
op reset_rx_ka_to : rx_ka_to -> Boolean
op reset_send_hello : send_hello -> Boolean
op reset_send_hold_to : send_hold_to -> Boolean
op reset_send_ka_to : send_ka_to -> Boolean
op reset_txKA : tx_KA -> Boolean
op reset_txinit : tx_init -> Boolean
op resetmode : mode -> Boolean
op rx_ack : FTP_tlv * ReCx_tlv * rs * seq * msg_q * Nat -> Boolean
op rx_hold_timer : rx_hold_to -> Boolean
op rx_msg : FTP_tlv * ReCx_tlv * rr * ack_seq_no * msg_q * Nat -> Boolean
op rxch : msg_q -> Boolean
op send : msg * address * interface -> Boolean
op sendLog :
```

```
    FTP_tlv *
    ReCx_tlv *
    chkpt_tlv *
    tcp_cx_failure *
    msg *
    address *
    interface *
    state *
    hello_adj_list -> Boolean
op send_ack :
    FTP_tlv *
    ReCx_tlv *
    rr *
    ack_seq_no *
    tcp_cx_failure *
    msg *
    address *
    interface *
    state *
    hello_adj_list -> Boolean
op send_hold_timer : send_hold_to -> Boolean
op send_msg :
    FTP_tlv *
    ReCx_tlv *
    rs *
    seq *
    tcp_cx_failure *
    msg *
    address *
    interface *
    state *
    hello_adj_list -> Boolean
op setNR : ack_seq_no -> Boolean
op setNS : seq -> Boolean
op setRxAck : rs -> Boolean
op setState : state * Nat -> Boolean
op setTxMsg : rr -> Boolean
op set_send_hello : send_hello -> Boolean
op set_txKA : tx_KA -> Boolean
op set_txinit : tx_init -> Boolean
op setmode : mode -> Boolean
op startReCxTimer : recx_to -> Boolean
```

```
op txch : msg_q -> Boolean
op writeLog : msg -> Boolean
axiom send_msg is
   fa(ftlv : FTP_tlv, rtlv : ReCx_tlv, r : rs, ns : seq, tcp : tcp_cx_failure,
      m : msg, a : address, i : interface, s : state, hl : hello_adj_list)
    ((ftlv = true) & ((rtlv = true) & (r = true))) =>
    ((resetRxAck r & mod_send(ftlv, tcp, m, a, i, s, hl, a)) => appendtxch ns)
axiom rx_ack is
   fa(ftlv : FTP_tlv, rtlv : ReCx_tlv, r : rs, ns : seq, rx_q : msg_q, n : Nat)
    ((ftlv = true) & ((rtlv = true) & ~(getrxch rx_q))) =>
    (setRxAck r & (assignrxch(nth(rx_q, n)) & setNS(ns + 1)))
axiom rx_msg is
   fa(ftlv : FTP_tlv, rtlv : ReCx_tlv, r : rr, nr : ack_seq_no,
      tx_q : msg_q, n : Nat)
    ((ftlv = true) & ((rtlv = true) & ~(gettxch tx_q))) =>
    (setTxMsg r & (assigntxch(nth(tx_q, n)) & setNR(getAckSeq(hd tx_q))))
axiom send_ack is
   fa(ftlv : FTP_tlv, rtlv : ReCx_tlv, r : rr, nr : ack_seq_no,
      tcp : tcp_cx_failure, m : msg, a : address, i : interface,
      s : state, hl : hello_adj_list)
    ((ftlv = true) & ((rtlv = true) & (r = true))) =>
    ((resetTxMsg r & mod_send(ftlv, tcp, m, a, i, s, hl, a)) => appendrxch nr)
axiom Reissue is
   fa(ftlv : FTP_tlv, rtlv : ReCx_tlv, r : rr, r : rs, ns : seq,
      tx_q : msg_q, rx_q : msg_q, my : my_ldp_id)

    ((ftlv = true) &
    ((rtlv = true) &
    (gettxch tx_q & (getrxch rx_q & ((r = false) & (r = false))))))) =>
    (ReInitialize my & appendtxch ns)
axiom ft_ldp_state_mc is
   fa(m : msg, s : state, sh : send_hello, myid : my_ldp_id, txi : tx_init,
      txKA : tx_KA, peer : peer_ldp_id, addr : address, self : address,
      hl : hello_adj_list, ifc : interface, ftlv : FTP_tlv,
      recx : ReCx_tlv, ckpt : chkpt_tlv, rec : record)
    if (s = 4) & ((ftlv = true) & ((ckpt = true) & ((recx = true) & isKA m)))
       then chkpoint(ftlv, recx, ckpt, rec)
    else ldp_state_mc(m, s, sh, myid, txi, txKA, peer, addr, self, hl, ifc)
axiom FT_RX_KA_TO is
   fa(m : msg, s : state, to : rx_ka_to, pkt : no_of_pkts_rxed,
      cx : tcp_cx_failure, hl : hello_adj_list, a : address, ft : FTP_tlv,
      re : ReCx_tlv, ck : chkpt_tlv, rec : record)
```

```
    RX_KA_TO1(ft, m, s, to, pkt, cx, hl, a) & chkpoint(ft, re, ck, rec)
axiom ReInit is
   fa(ft : FTP_tlv, re : ReCx_tlv, ck : chkpt_tlv, lka : last_kaseq_no,
      tcp : tcp_cx_failure, ka : kaseq_no, m : msg, a : address,
      i : interface, s : state, hl : hello_adj_list, recx : recx_to,
      rxho : rx_hold_to)
    if (lka < ka) & (tcp = true)
        then
       startReCxTimer recx & (reset_rx_hold_to rxho & Recover(ft, re))

    else sendLog(ft, re, ck, tcp, m, a, i, s, hl)
axiom chkpoint is
   fa(ft : FTP_tlv, re : ReCx_tlv, ck : chkpt_tlv, rec : record)
    ((ft = true) & ((re = true) & (ck = true))) => flush rec
axiom sendLog is
   fa(ft : FTP_tlv, re : ReCx_tlv, ck : chkpt_tlv, tcp : tcp_cx_failure,
      m : msg, a : address, i : interface, s : state, hl : hello_adj_list)
    if ft & (re & ck)
        then writeLog m & mod_send(ft, tcp, m, a, i, s, hl, a)
    else mod_send(ft, tcp, m, a, i, s, hl, a)
axiom mod_send is
   fa(ftlv : FTP_tlv, cx : tcp_cx_failure, m : msg, a : address,
      ifc : interface, s : state, hl : hello_adj_list, peer : address)
    ((ftlv = true) & (cx = true)) =>
    ((s = 4) &
    (hello_adj(hl, peer) &
    ((send(m, a, ifc) = false) & (receive(m, peer, ifc) = false))))
axiom RX_KA_TO1 is
   fa(ftlv : FTP_tlv, m : msg, s : state, to : rx_ka_to,
      pkt : no_of_pkts_rxed, cx : tcp_cx_failure, hl : hello_adj_list,
      a : address)

    ((ftlv = true) &
    (RX_KA_TO(m, s, to, pkt) & ((s = 4) & (hello_adj(hl, a) & (pkt = 0)))))
    => (reset_rx_ka_to to & (cx = true))
axiom create_hello_adj is
   fa(hl : hello_adj_list, a : address) hd(insert(a, hl)) = a
axiom hello_adj is
   fa(hl : hello_adj_list, a : address) (~(hl = []) & member(a, hl)) => true
axiom interface_config is
   fa(il : if_config_list, i : interface)
    (~(il = []) & member(i, il)) => true
```

```
axiom peer_allowed is
   fa(ap : allowed_peers, a : address) (~(ap = []) & member(a, ap)) => true
axiom receive is
   fa(m : msg, a : address, ifc : interface)
    ~(send(m, a, ifc)) & receive(m, a, ifc)
axiom send is
   fa(m : msg, a : address, ifc : interface)
    ~(receive(m, a, ifc)) & send(m, a, ifc)
axiom hello_acceptable is
   fa(il : if_config_list, ifc : interface, ok : hello_ok, a : address,
      ap : allowed_peers)
    (interface_config(il, ifc) & peer_allowed(ap, a)) => (ok = true)
axiom ldp_start is
   fa(b : send_hello, H : msg, p : address, ifc : interface)
    ((b = true) & ~(p = 0)) => (send(H, p, ifc) & receive(H, p, ifc))
axiom SEND_HELLO is
   fa(b : send_hello, m : msg, a : address, ifc : interface)
    ((b = true) & send(m, a, ifc)) => reset_send_hello b
axiom RECEIVE_HELLO is
   fa(hl : hello_adj_list, b : send_hello, s : address, p : address,
      i : interface, ok : hello_ok, m : mode, ms : msg,
      sho : send_hold_to, rho : rx_hold_to)

    (receive(ms, p, i) &
    (reset_rx_hold_to rho &
    if hello_adj(hl, p)
        then
       if s > p
           then setmode m & TCP_CX(s, p)
       else resetmode m & TCP_ACCEPT(s, p)

    else

    (create_hello_adj(hl, p) &
    (set_send_hello b &
    if s > p
        then setmode m & TCP_CX(s, p)
    else resetmode m & TCP_ACCEPT(s, p))) => SEND_HELLO(b, ms, p, i)))
    => rx_hold_timer rho
axiom ldp_Init is
   fa(I : msg, idc : my_ldp_id, ids : peer_ldp_id, tx : tx_init,
   s : state, a : address, i : interface)
```

```
    ((s = 1) & set_txinit tx) =>
    (SEND_INIT(I, tx, idc, ids, a, i) & setState(s, 2))
axiom SEND_INIT is
   fa(h : msg, b : tx_init, idc : my_ldp_id, ids : peer_ldp_id, a : address,
      ifc : interface) (b = true) => (send(h, a, ifc) & reset_txinit b)
axiom RECEIVE_INITcs is
   fa(I : msg, N : msg, K : msg, id : rxr_ldp_id, s : state,
   idc : my_ldp_id, ka : tx_KA, a : address, i : interface)
    if (s = 2) & (id = idc)
        then set_txKA ka => (SEND_KA(K, ka, a, i) & setState(s, 3))
    else SEND_NOTFN(N, a) & setState(s, 5)
axiom SEND_KA is
   fa(K : msg, b : tx_KA, a : address, i : interface)
    ((b = true) & send(K, a, i)) => reset_txKA b
axiom RECEIVE_KA is fa(k : msg, s : state) (s = 3) => setState(s, 4)
axiom RX_KA_TO is
   fa(K : msg, s : state, to : rx_ka_to, pkt : no_of_pkts_rxed)
    ((to = true) & (pkt = 0)) => (setState(s, 5) & reset_rx_ka_to to)
axiom SEND_KA_TO is
   fa(K : msg, s : state, to : send_ka_to, pkt : no_of_pkts_sent,
      tx : tx_KA, a : address, i : interface)
    ((pkt = 0) & (to = true)) => (set_txKA tx & (SEND_KA(K, tx, a, i)
    & (reset_send_ka_to to & (pkt = 0))))
axiom SEND_HOLD_TO is
   fa(H : msg, hto : send_hold_to, tx : send_hello, a : address, i : interface)
    (hto = true) => (set_send_hello tx &
    (SEND_HELLO(tx, H, a, i) & reset_send_hold_to hto))
axiom RX_HOLD_TO is
   fa(H : msg, to : rx_hold_to) (to = true) => reset_rx_hold_to to
axiom ldp_state_mc is
   fa(m : msg, s : state, sh : send_hello, myid : my_ldp_id,
      txi : tx_init, txKA : tx_KA, peer : peer_ldp_id, addr : address,
      self : address, hl : hello_adj_list, ifc : interface)

    if (s = 5) & isHello m
        then SEND_HELLO(sh, m, addr, ifc)
    else
    if (s = 5) & isConnect m
        then SEND_INIT(m, txi, myid, peer, addr, ifc)
    else SEND_NOTFN(m, addr) & close myid or
    (
    if (s = 1) & isHello m
```

```
        then SEND_HELLO(sh, m, addr, ifc)
    else
    if (s = 1) & isInit m
        then SEND_INIT(m, txi, myid, peer, addr, ifc)
    else SEND_NOTFN(m, addr) & close myid or
    (
    if (s = 2) & isHello m
        then SEND_HELLO(sh, m, addr, ifc)
    else
    if (s = 2) & isInit m
        then SEND_INIT(m, txi, myid, peer, addr, ifc)
    else SEND_NOTFN(m, addr) & close myid or
    (
    if (s = 3) & isKA m
        then EXCG_ADDR_LBL_MSG(m, myid, peer, txi, s, addr, self, ifc, hl, m)
    else SEND_NOTFN(m, addr) & close myid or
    if (s = 4) & isKA m
        then SEND_HELLO(sh, m, addr, ifc)
    else
    if (s = 4) & (isInit m or isKA m)
        then SEND_KA(m, txKA, addr, ifc)
    else SEND_NOTFN(m, addr) & close myid)))
axiom EXCG_ADDR_LBL_MSG is
   fa(I : msg, idc : my_ldp_id, ids : peer_ldp_id, tx : tx_init,
      s : state, peer : address, se : address, ifc : interface,
      hl : hello_adj_list, m : msg)

    (ldp_Init(I, idc, ids, tx, s, peer, ifc) &
    (hello_adj(hl, peer) & (SEND_ADDR(m, peer) or SEND_LBL(m, peer))))
    => ((RECEIVE_ADDR(m, se) or RECEIVE_LBL(m, se)) & (s = 4))
endspec

Processing prove at Z:/Win2K_Profile/ldp/ftldp#p5
Processing spec at C:/Program Files/Specware4.0/Library/Base/ProverBase
p5: Theorem FT_EXCG_ADDR_LBL_MSG in ft_ldp is Proved!
    Snark Log file: Z:/Win2K_Profile/ldp/snark/ftldp/p5.log
CL-USER(3):
```

Bibliography

[1] L. Andersson, P. Doolan, N. Feldman, A. Fredette and B. Thomas, "LDP Specification", *RFC3036.*, Jan 2001.

[2] A. Arora, *Efficient Reconfiguration of Trees: A Case Study in Methodical Design of Nonmasking Fault-Tolerant Programs*, Science of Computer Programming, 1996.

[3] A. Arora and M. G. Gouda, "Closure and Convergence: A Foundation of Fault-Tolerant Computing,". *IEEE Transactions on Software Engineering*, **19(11)**, (1993) 1015-1027.

[4] A. Arora and S. Kulkarni, "Component Based Design of Multitolerant Systems", *IEEE Transactions on Software Engineering*, vol. 24(1), pp. 63-78, January. 1998.

[5] A. Arora and S. Kulkarni, "Designing Masking Fault-tolerance via Nonmasking Fault-tolerance," *IEEE Transactions on Software Engineering*, **24(6)**, (1998) 435-450.

[6] F. Babich and L. Deotto, "Formal Methods for Specification and Analysis of Communication Protocols", *IEEE Communications Surveys and Tutorials.*, Dec 2002.

[7] Grady Booch, "Software Components with Ada : Structures, Tools, and Subsystems," (Benjamin/Cummings Pub. Co., 1987)

[8] A.W. Brown, K. C. Wallnau, "The Current State of CBSE," *IEEE Software*, (1998) 37-46.

[9] K.M.Chandy and J.Misra, *Parallel Program Design: A Foundation.*, (Addison-Wesley, 1988).

[10] Paul C. Clements: "From Subroutines to Subsystems: Component-Based Software Development", Software Engineering Institute, Carnegie Mellon University (Pittsburgh, Pennsylvania).

[11] Neville Dean, *The Essence of Discrete Mathematics*, (Prentice Hall, 1997).

[12] N. Delisle and D. Garlan, "Formally Specifying Electronic Instruments",*Fifth International Workshop on Software Specification and Design*, pp.242-248, 1989.

[13] S.H. Edwards, D.S. Gibson, B.W. Weide, S. Zhupanov, "Software Component Relationships," http://www.umcs.maine.edu/ftp/wisr/wisr8/papers/edwards/edwards.html

[14] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 2, Module Spec. and Constraints*, (Springer-Verlag, Berlin 1990).

[15] A. Farrel, "Fault Tolerance for the Label Distribution Protocol", *RFC 3479.*, Feb 2003.

[16] Felix C. Gartner: "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments".

[17] C. Michael Holloway, "Software Engineering and Epistemology",*Software Engineering Notes*, vol 20(2), pp. 20, April 1995.

[18] Kestrel Institute, "Specware 4.0 Tutorial," http://www.specware.org/documentation/4.0/tutorial/SpecwareTutorial.html.

[19] S. Kulkarni: "Component Based Design of Fault-Tolerance",*Ph.D Dissertation*, The Ohio State University, 1999.

[20] S.S. Kulkarni and A. Arora, "Once-and-Forall Management Protocol(OFMP)", *Proceedings of the Fifth International Conference on Network Protocols.*, Oct 1997.

[21] Z.Liu, M.Joseph, "Transformation of Programs for Fault-Tolerance," *Formal Aspects of Computing*, **4(5)**, (1992) 442-469.

[22] Z. Liu, M. Joseph, "Verification of Fault Tolerance and Real Time," *FTCS-26*, (Sendai, Japan, 1996), 220-229.

[23] David McKinley: "Open Fault Tolerance through System Directed Checkpointing," White Paper, RadiSys Corporation, Nov 1999.

[24] K. L. McMillan, "Fitting Formal Methods into the Design Cycle," *Proc. 31st Design Automation Conference*, (1994) pp. 314-19.

[25] Pierre Michel and Virginie Wiels, "A Framework for Modular Formal Specification and Verification",*Proceedings of FME'97*, no. 1313 in LNCS, Springer-Verlag, 1997.

[26] C. J. Nix and B. P. Collins, "The Use of Software Engineering, including the Z Notation, in the Development of CICS". Quality Assurance 74 (September 1988).

[27] L.L. Pullum, *Software Fault-tolerance Techniques and Implementation*, (Artech House, Boston, 2001).

[28] K. Raymond, "A Tree Based Algorithm for Mutual Exclusion," *ACM Transactions on Computer Systems*, **7**, (1989) 61–77.

[29] E. Rosen, A. Viswanathan and R. Callon, "Multiprotocol Label Switching Architecture", *RFC3031.*, Jan 2001.

[30] F.B. Schneider, "Implementing Fault Tolerant Services using State Machine Approach : A tutorial," *ACM Computing Surveys* **22(4)**, (1990) 299-319.

[31] J.L.A. van de Snepscheut, "Fair Mutual Exclusion on a Graph of Processes," *Distributed Computing*, **2(2)**, (1987) 113-115.

[32] Y. V. Srinivas, "Category Theory Definitions and Examples," Technical Report, Dept of Information and Computer Science, University of California, Irvine, Feb 1990. TR 90-14.

[33] Y.V. Srinivas and R. Jullig, "Specware(TM): Formal Support for Composing Software", *Proc. of the Conference on Mathematics of Program Construction.*, B.Moeller, Ed. LNCS 947, Springer-Verlag, pp. 399-422, 1995.

[34] Y. V. Srinivas and J. L. McDonald, "The Architecture of SPECWARE, a Formal Software Development System," Technical Report, Kestrel Institute, 1996.

[35] J. Voas, "Certifying Off-the-Shelf Software Components," *IEEE Computer*, **31(6)**, (1998) 53-59.

[36] William G. Wood, "Application of Formal Methods to System and Software Specification", Software Engineering Institute. Department of Carnegie Mellon University, Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development. Napa, California, May 9-11, 1990.

Publications

- A. Hanumantharaya, P. Sinha, "On the Use of Category Theory for Component-Based Fault-Tolerant Software Development," Proc. of International Conference on Information Technology, India, Dec. 2002 (Awarded *"the Best Paper Award"*).
- A. Hanumantharaya, P. Sinha, A. Agarwal, "A Component-Based Design of a Fault-Tolerant Multimedia Communication Protocol," Proc. of IEEE Multimedia Software Engineering (MSE 2003), Taiwan, Dec. 2003.
- A. Hanumantharaya, P. Sinha, A. Agarwal, "A Component-Based Design and Compositional Verification of a Fault-Tolerant Multimedia Communication Protocol," a special issue on Software Engineering in the Journal of Real-time Imaging, Academic Press, July 2003.